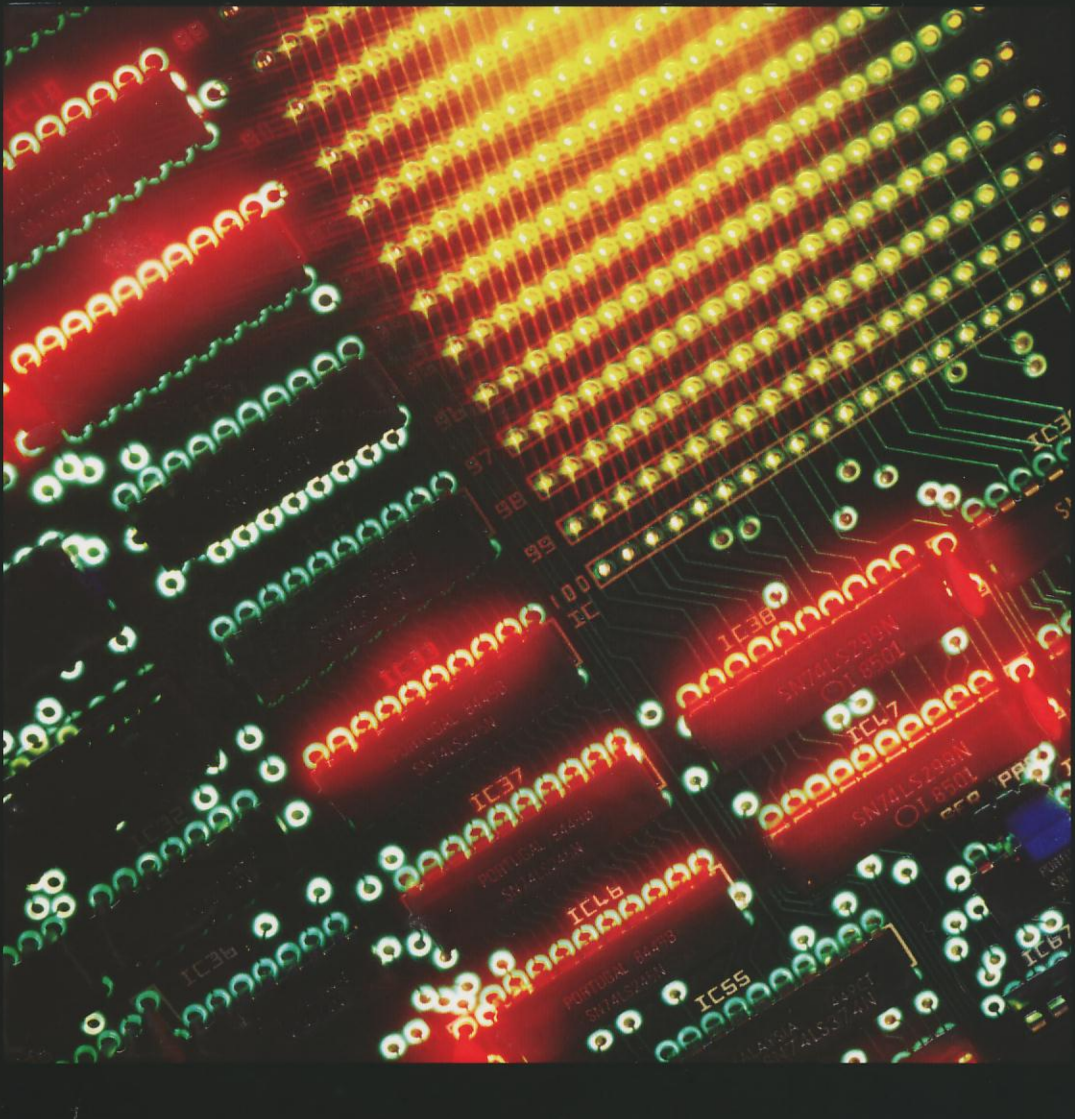
TENTH EDITION

10th Anniversary Edition

www.prenhall.com

DIGITAL SYSTEMS

PRINCIPLES AND APPLICATIONS



SOFTWARE ENCLOSED

RONALD J. TOCCI

NEAL S. WIDMER

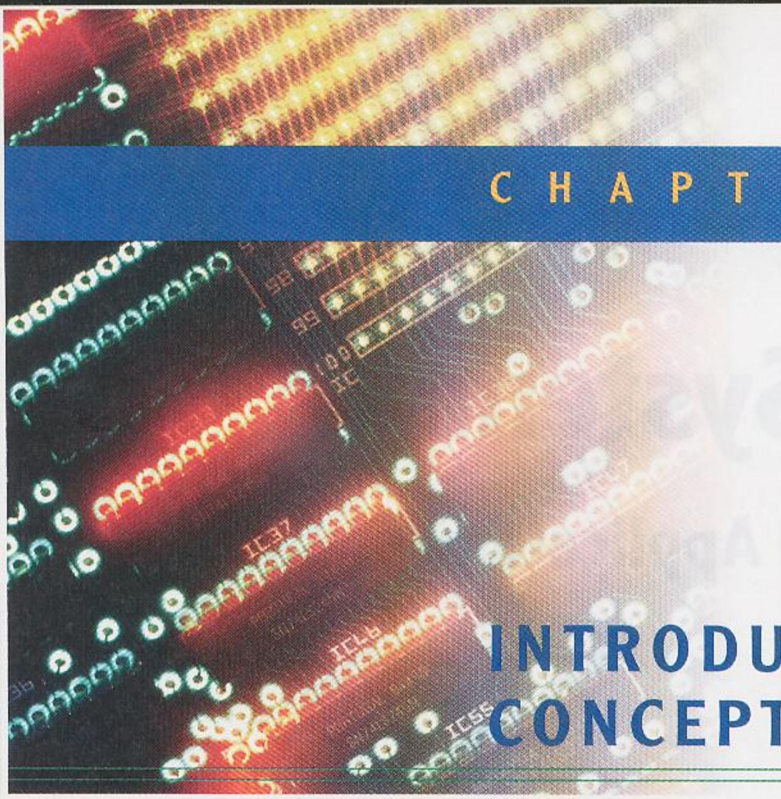
GREGORY L. MOSS

VISIT…

LANZAROTE
Caliente.COM

CHAPTER 1

INTRODUCTORY CONCEPTS



OUTLINE

1-1 Numerical Representations
1-2 Digital and Analog Systems
1-3 Digital Number Systems
1-4 Representing Binary Quantities
1-5 Digital Circuits/Logic Circuits
1-6 Parallel and Serial Transmission
1-7 Memory
1-8 Digital Computers

OBJECTIVES

Upon completion of this chapter, you will be able to:

- Distinguish between analog and digital representations.
- Cite the advantages and drawbacks of digital techniques compared with analog.
- Understand the need for analog-to-digital converters (ADCs) and digital-to-analog converters (DACs).
- Recognize the basic characteristics of the binary number system.
- Convert a binary number to its decimal equivalent.
- Count in the binary number system.
- Identify typical digital signals.
- Identify a timing diagram.
- State the differences between parallel and serial transmission.
- Describe the property of memory.
- Describe the major parts of a digital computer and understand their functions.
- Distinguish among microcomputers, microprocessors, and microcontrollers.

INTRODUCTION

In today's world, the term *digital* has become part of our everyday vocabulary because of the dramatic way that digital circuits and digital techniques have become so widely used in almost all areas of life: computers, automation, robots, medical science and technology, transportation, telecommunications, entertainment, space exploration, and on and on. You are about to begin an exciting educational journey in which you will discover the fundamental principles, concepts, and operations that are common to all digital systems, from the simplest on/off switch to the most complex computer. If this book is successful, you should gain a deep understanding of how all digital systems work, and you should be able to apply this understanding to the analysis and troubleshooting of any digital system.

We start by introducing some underlying concepts that are a vital part of digital technology; these concepts will be expanded on as they are needed later in the book. We also introduce some of the terminology that is necessary when embarking on a new field of study, and add to this list of important terms in every chapter.

1-1 NUMERICAL REPRESENTATIONS

In science, technology, business, and, in fact, most other fields of endeavor, we are constantly dealing with *quantities*. Quantities are measured, monitored, recorded, manipulated arithmetically, observed, or in some other way utilized in most physical systems. It is important when dealing with various quantities that we be able to represent their values efficiently and accurately. There are basically two ways of representing the numerical value of quantities: **analog** and **digital**.

Analog Representations

In **analog representation** a quantity is represented by a continuously variable, proportional indicator. An example is an automobile speedometer from the classic muscle cars of the 1960s and 1970s. The deflection of the needle is proportional to the speed of the car and follows any changes that occur as the vehicle speeds up or slows down. On older cars, a flexible mechanical shaft connected the transmission to the speedometer on the dash board. It is interesting to note that on newer cars, the analog representation is usually preferred even though speed is now measured digitally.

Thermometers before the digital revolution used analog representation to measure temperature, and many are still in use today. Mercury thermometers use a column of mercury whose height is proportional to temperature. These devices are being phased out of the market because of environmental concerns, but nonetheless they are an excellent example of analog representation. Another example is an outdoor thermometer on which the position of the pointer rotates around a dial as a metal coil expands and contracts with temperature changes. The position of the pointer is proportional to the temperature. Regardless of how small the change in temperature, there will be a proportional change in the indication.

In these two examples the physical quantities (speed and temperature) are being coupled to an indicator by purely mechanical means. In electrical analog systems, the physical quantity that is being measured or processed is converted to a proportional voltage or current (electrical signal). This voltage or current is then used by the system for display, processing, or control purposes.

Sound is an example of a physical quantity that can be represented by an electrical analog signal. A microphone is a device that generates an output voltage that is proportional to the amplitude of the sound waves that strike it. Variations in the sound waves will produce variations in the microphone's output voltage. Tape recordings can then store sound waves by using the output voltage of the microphone to proportionally change the magnetic field on the tape.

Analog quantities such as those cited above have an important characteristic, no matter how they are represented: *they can vary over a continuous range of values*. The automobile speed can have *any* value between zero and, say, 100 mph. Similarly, the microphone output might have any value within a range of zero to 10 mV (e.g., 1 mV, 2.3724 mV, 9.9999 mV).

Digital Representations

In **digital representation** the quantities are represented not by continuously variable indicators but by symbols called *digits*. As an example, consider the digital clock, which provides the time of day in the form of decimal digits that represent hours and minutes (and sometimes seconds). As we know, the time of day changes continuously, but the digital clock reading does not change continuously; rather, it changes in steps of one per minute (or per second). In other words, this digital representation of the time of day changes in *discrete* steps, as compared with the representation of time provided by an analog ac line-powered wall clock, where the dial reading changes continuously.

The major difference between analog and digital quantities, then, can be simply stated as follows:

analog ≡ continuous
digital ≡ discrete (step by step)

Because of the discrete nature of digital representations, there is no ambiguity when reading the value of a digital quantity, whereas the value of an analog quantity is often open to interpretation. In practice, when we take a measurement of an analog quantity, we always "round" to a convenient level of precision. In other words, we digitize the quantity. The digital representation is the result of assigning a number of limited precision to a continuously variable quantity. For example, when you take your temperature with a mercury (analog) thermometer, the mercury column is usually between two graduation lines, but you would pick the nearest line and assign it a number of, say, 98.6°F.

EXAMPLE 1-1

Which of the following involve analog quantities and which involve digital quantities?

(a) Ten-position switch
(b) Current flowing from an electrical outlet
(c) Temperature of a room
(d) Sand grains on the beach
(e) Automobile fuel gauge

Solution

(a) Digital
(b) Analog
(c) Analog
(d) Digital, since the number of grains can be only certain discrete (integer) values and not every possible value over a continuous range
(e) Analog, if needle type; digital, if numerical readout or bar graph display

REVIEW QUESTION *

1. Concisely describe the major difference between analog and digital quantities.

1-2 DIGITAL AND ANALOG SYSTEMS

A **digital system** is a combination of devices designed to manipulate logical information or physical quantities that are represented in digital form; that is, the quantities can take on only discrete values. These devices are most often electronic, but they can also be mechanical, magnetic, or pneumatic. Some of the more familiar digital systems include digital computers and calculators, digital audio and video equipment, and the telephone system—the world's largest digital system.

---

*Answers to review questions are found at the end of the chapter in which they occur.

An **analog system** contains devices that manipulate physical quantities that are represented in analog form. In an analog system, the quantities can vary over a continuous range of values. For example, the amplitude of the output signal to the speaker in a radio receiver can have any value between zero and its maximum limit. Other common analog systems are audio amplifiers, magnetic tape recording and playback equipment, and a simple light dimmer switch.

Advantages of Digital Techniques

An increasing majority of applications in electronics, as well as in most other technologies, use digital techniques to perform operations that were once performed using analog methods. The chief reasons for the shift to digital technology are:

1. *Digital systems are generally easier to design.* The circuits used in digital systems are *switching circuits,* where *exact* values of voltage or current are not important, only the range (HIGH or LOW) in which they fall.
2. *Information storage is easy.* This is accomplished by special devices and circuits that can latch onto digital information and hold it for as long as necessary, and mass storage techniques that can store billions of bits of information in a relatively small physical space. Analog storage capabilities are, by contrast, extremely limited.
3. *Accuracy and precision are easier to maintain throughout the system.* Once a signal is digitized, the information it contains does not deteriorate as it is processed. In analog systems, the voltage and current signals tend to be distorted by the effects of temperature, humidity, and component tolerance variations in the circuits that process the signal.
4. *Operation can be programmed.* It is fairly easy to design digital systems whose operation is controlled by a set of stored instructions called a *program.* Analog systems can also be *programmed,* but the variety and the complexity of the available operations are severely limited.
5. *Digital circuits are less affected by noise.* Spurious fluctuations in voltage (noise) are not as critical in digital systems because the exact value of a voltage is not important, as long as the noise is not large enough to prevent us from distinguishing a HIGH from a LOW.
6. *More digital circuitry can be fabricated on IC chips.* It is true that analog circuitry has also benefited from the tremendous development of IC technology, but its relative complexity and its use of devices that cannot be economically integrated (high-value capacitors, precision resistors, inductors, transformers) have prevented analog systems from achieving the same high degree of integration.

Limitations of Digital Techniques

There are really very few drawbacks when using digital techniques. The two biggest problems are:

The real world is analog.
Processing digitized signals takes time.

Most physical quantities are analog in nature, and these quantities are often the inputs and outputs that are being monitored, operated on, and controlled by a system. Some examples are temperature, pressure, position, velocity, liquid level, flow rate, and so on. We are in the habit of expressing these quantities *digitally,* such as when we say that the temperature is 64° (63.8° when we want to be more precise), but we are really making a digital approximation to an inherently analog quantity.

To take advantage of digital techniques when dealing with analog inputs and outputs, four steps must be followed:

1. Convert the physical variable to an electrical signal (analog).
2. Convert the electrical (analog) signal into digital form.
3. Process (operate on) the digital information.
4. Convert the digital outputs back to real-world analog form.

An entire book could be written about step 1 alone. There are many kinds of devices that convert various physical variables into electrical analog signals (sensors). These are used to measure things that are found in our "real" analog world. On your car alone, there are sensors for fluid level (gas tank), temperature (climate control and engine), velocity (speedometer), acceleration (airbag collision detection), pressure (oil, manifold), and flow rate (fuel), to name just a few.

To illustrate a typical system that uses this approach Figure 1-1 describes a precision temperature regulation system. A user pushes up or down buttons to set the desired temperature in 0.1° increments (digital representation). A temperature sensor in the heated space converts the measured temperature to a proportional voltage. This analog voltage is converted to a digital quantity by an **analog-to-digital converter (ADC)**. This value is then compared to the desired value and used to determine a digital value of how much heat is needed. The digital value is converted to an analog quantity (voltage) by a **digital-to-analog converter (DAC)**. This voltage is applied to a heating element, which will produce heat that is related to the voltage applied and will affect the temperature of the space.

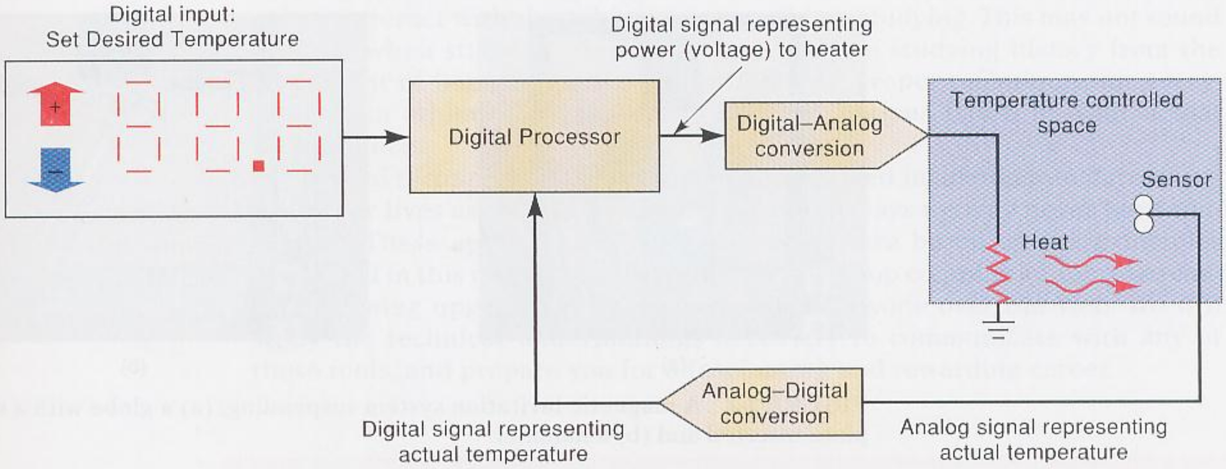

Digital input:
Set Desired Temperature
+
–
Digital Processor
Digital signal representing power (voltage) to heater
Digital–Analog conversion
Temperature controlled space
Sensor
Heat
Analog–Digital conversion
Digital signal representing actual temperature
Analog signal representing actual temperature

FIGURE 1-1 Block diagram of a precision digital temperature control system.

Another good example where conversion between analog and digital takes place is in the recording of audio. Compact disks (CDs) have replaced cassette tapes because they provide a much better means for recording and playing back music. The process works something like this: (1) sounds from instruments and human voices produce an analog voltage signal in a microphone; (2) this analog signal is converted to a digital format using an analog-to-digital conversion process; (3) the digital information is stored on the CD's surface; (4) during playback, the CD player takes the digital information from the CD surface and converts it into an analog signal that is then amplified and fed to a speaker, where it can be picked up by the human ear.

The second drawback to digital systems is that processing these digitized signals (lists of numbers) takes time. And we also need to convert between the analog and digital forms of information, which can add complexity and expense to a system. The more precise the numbers need to be, the longer it takes to process them. In many applications, these factors are outweighed by the numerous advantages of using digital techniques, and so the conversion between analog and digital quantities has become quite commonplace in the current technology.

There are situations, however, where use of analog techniques is simpler or more economical. For example, several years ago, a colleague (Tom Robertson) decided to create a control system demonstration for tour groups. He planned to suspend a metallic object in a magnetic field, as shown in Figure 1-2. An electromagnet was made by winding a coil of wire and controlling the amount of current through the coil. The position of the metal object was measured by passing an infrared light beam across the magnetic field. As the object drew closer to the magnetic coil, it began to block the light beam. By measuring small changes in the light level, the magnetic field could be controlled to keep the metal object hovering and stationary, with no strings attached. All attempts at using a microcomputer to measure these very small changes, run the control calculations, and drive the magnet proved to be too slow, even when using the fastest, most powerful PC available at the time. His final solution used just a couple of op-amps and a few dollars' worth of other components: a totally analog approach. Today we have access to processors fast enough and measurement techniques precise enough to accomplish this feat, but the simplest solution is still analog.

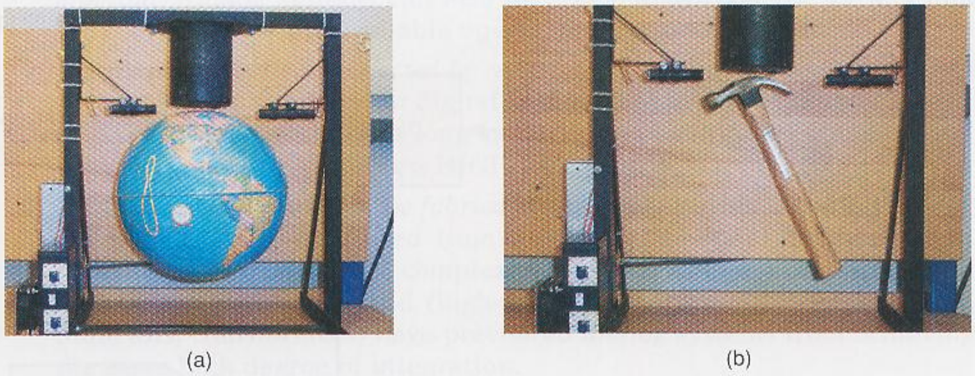

(a) (b)

FIGURE 1-2 A magnetic levitation system suspending: (a) a globe with a steel plate inserted and (b) a hammer.

It is common to see both digital and analog techniques employed within the same system to be able to profit from the advantages of each. In these *hybrid* systems, one of the most important parts of the design phase involves determining what parts of the system are to be analog and what parts are to be digital. The trend in most systems is to digitize the signal as early as possible and convert it back to analog as late as possible as the signals flow through the system.

The Future Is Digital

The advances in digital technology over the past three decades have been nothing short of phenomenal, and there is every reason to believe that more is coming. Think of the everyday items that have changed from analog format to digital in your lifetime. An indoor/outdoor wireless digital thermometer can be purchased for less then $10.00. Cars have gone from having very few electronic controls to being predominantly digitally controlled vehicles. Digital audio has moved us to the compact disk and MP3 player. Digital video brought the DVD. Digital home video and still cameras; digital recording with systems like TiVo; digital cellular phones; and digital imaging in x-ray, magnetic resonance imaging (MRI), and ultrasound systems in hospitals are just a few of the applications that have been taken over by the digital revolution. As soon as the infrastructure is in place, telephone and television systems will go digital. The growth rate in the digital realm continues to be staggering. Maybe your automobile is equipped with a system such as GM's On Star, which turns your dashboard into a hub for wireless communication, information, and navigation. You may already be using voice commands to send or retrieve e-mail, call for a traffic report, check on the car's maintenance needs, or just switch radio stations or CDs—all without taking your hands off the wheel or your eyes off the road. Cars can report their exact location in case of emergency or mechanical breakdown. In the coming years wireless communication will continue to expand coverage to provide connectivity wherever you are. Telephones will be able to receive, sort, and maybe respond to incoming calls like a well-trained secretary. The digital television revolution will provide not only higher definition of the picture, but also much more flexibility in programming. You will be able to select the programs that you want to view and load them into your television's memory, allowing you to pause or replay scenes at your convenience, very much like viewing a DVD today. As virtual reality continues to improve, you will be able to interact with the subject matter you are studying. This may not sound exciting when studying electronics, but imagine studying history from the standpoint of being a participant, or learning proper techniques for everything from athletics to surgery through simulations based on your actual performance.

Digital technology will continue its high-speed incursion into current areas of our lives as well as break new ground in ways we may never have considered. These applications (and many more) are based on the principles presented in this text. The software tools to develop complex systems are constantly being upgraded and are available to anyone over the Web. We will study the technical underpinnings necessary to communicate with any of these tools, and prepare you for a fascinating and rewarding career.

REVIEW QUESTIONS

1. What are the advantages of digital techniques over analog?
2. What is the chief limitation to the use of digital techniques?

1-3 DIGITAL NUMBER SYSTEMS

Many number systems are in use in digital technology. The most common are the decimal, binary, octal, and hexadecimal systems. The decimal system is clearly the most familiar to us because it is a tool that we use every day. Examining some of its characteristics will help us to understand the other systems better.

Decimal System

The **decimal system** is composed of *10* numerals or symbols. These 10 symbols are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9; using these symbols as *digits* of a number, we can express any quantity. The decimal system, also called the *base-10* system because it has 10 digits, has evolved naturally as a result of the fact that people have 10 fingers. In fact, the word *digit* is derived from the Latin word for "finger."

The decimal system is a *positional-value system* in which the value of a digit depends on its position. For example, consider the decimal number 453. We know that the digit 4 actually represents 4 *hundreds,* the 5 represents 5 *tens,* and the 3 represents 3 *units.* In essence, the 4 carries the most weight of the three digits; it is referred to as the *most significant digit (MSD).* The 3 carries the least weight and is called the *least significant digit (LSD).*

Consider another example, 27.35. This number is actually equal to 2 tens plus 7 units plus 3 tenths plus 5 hundredths, or $2 \times 10 + 7 \times 1 + 3 \times 0.1 + 5 \times 0.01$. The decimal point is used to separate the integer and fractional parts of the number.

More rigorously, the various positions relative to the decimal point carry weights that can be expressed as powers of 10. This is illustrated in Figure 1-3, where the number 2745.214 is represented. The decimal point separates the positive powers of 10 from the negative powers. The number 2745.214 is thus equal to

$$(2 \times 10^{+3}) + (7 \times 10^{+2}) + (4 \times 10^{1}) + (5 \times 10^{0}) + (2 \times 10^{-1}) + (1 \times 10^{-2}) + (4 \times 10^{-3})$$

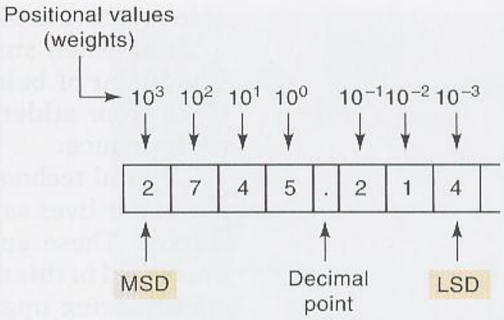


FIGURE 1-3 Decimal position values as powers of 10.

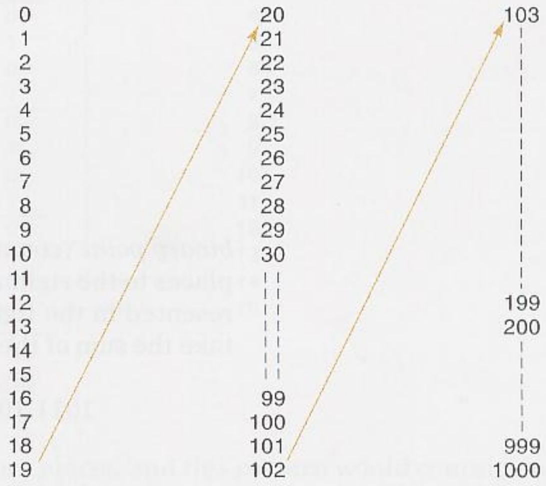
In general, any number is simply the sum of the products of each digit value and its positional value.

Decimal Counting

When counting in the decimal system, we start with 0 in the units position and take each symbol (digit) in progression until we reach 9. Then we add a 1 to the next higher position and start over with 0 in the first position (see

Figure 1-4). This process continues until the count of 99 is reached. Then we add a 1 to the third position and start over with 0s in the first two positions. The same pattern is followed continuously as high as we wish to count.

FIGURE 1-4 Decimal counting.



It is important to note that in decimal counting, the units position (LSD) changes upward with each step in the count, the tens position changes upward every 10 steps in the count, the hundreds position changes upward every 100 steps in the count, and so on.

Another characteristic of the decimal system is that using only two decimal places, we can count through $10^2 = 100$ different numbers (0 to 99).* With three places we can count through 1000 numbers (0 to 999), and so on. In general, with N places or digits, we can count through 10^N different numbers, starting with and including zero. The largest number will always be $10^N - 1$.

Binary System

Unfortunately, the decimal number system does not lend itself to convenient implementation in digital systems. For example, it is very difficult to design electronic equipment so that it can work with 10 different voltage levels (each one representing one decimal character, 0 through 9). On the other hand, it is very easy to design simple, accurate electronic circuits that operate with only two voltage levels. For this reason, almost every digital system uses the binary (base-2) number system as the basic number system of its operations. Other number systems are often used to interpret or represent binary quantities for the convenience of the people who work with and use these digital systems.

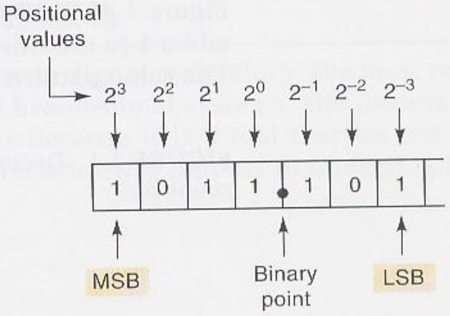
In the **binary system** there are only two symbols or possible digit values, 0 and 1. Even so, this base-2 system can be used to represent any quantity that can be represented in decimal or other number systems. In general though, it will take a greater number of binary digits to express a given quantity.

All of the statements made earlier concerning the decimal system are equally applicable to the binary system. The binary system is also a positional-value system, wherein each binary digit has its own value or weight expressed as a power of 2. This is illustrated in Figure 1-5. Here, places to the left of the

---

*Zero is counted as a number.

FIGURE 1-5 Binary position values as powers of 2.



binary point (counterpart of the decimal point) are positive powers of 2, and places to the right are negative powers of 2. The number 1011.101 is shown represented in the figure. To find its equivalent in the decimal system, we simply take the sum of the products of each digit value (0 or 1) and its positional value:

$$\begin{aligned} 1011.101_2 &= (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) \\ &\quad + (1 \times 2^{-1}) + (0 \times 2^{-2}) + (1 \times 2^{-3}) \\ &= 8 + 0 + 2 + 1 + 0.5 + 0 + 0.125 \\ &= 11.625_{10} \end{aligned}$$

Notice in the preceding operation that subscripts (2 and 10) were used to indicate the base in which the particular number is expressed. This convention is used to avoid confusion whenever more than one number system is being employed.

In the binary system, the term *binary digit* is often abbreviated to the term **bit**, which we will use from now on. Thus, in the number expressed in Figure 1-5 there are four bits to the left of the binary point, representing the integer part of the number, and three bits to the right of the binary point, representing the fractional part. The most significant bit (MSB) is the leftmost bit (largest weight). The least significant bit (LSB) is the rightmost bit (smallest weight). These are indicated in Figure 1-5. Here, the MSB has a weight of 2^3; the LSB has a weight of 2^{-3}.

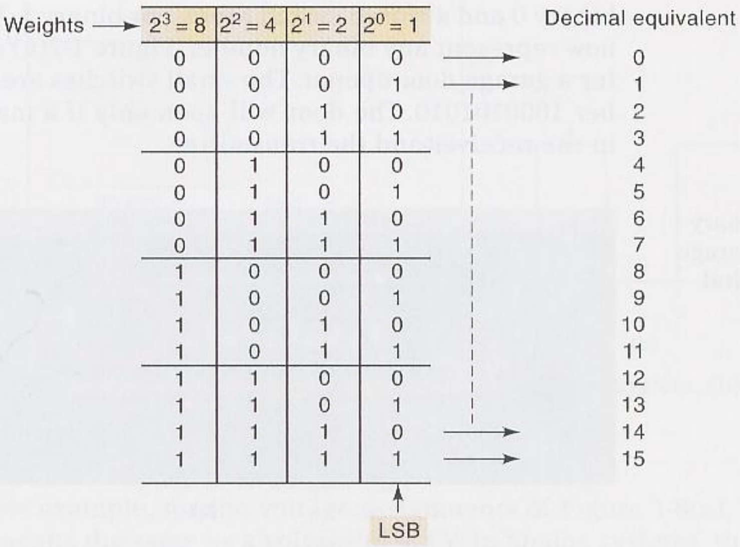
Binary Counting

When we deal with binary numbers, we will usually be restricted to a specific number of bits. This restriction is based on the circuitry used to represent these binary numbers. Let's use four-bit binary numbers to illustrate the method for counting in binary.

The sequence (shown in Figure 1-6) begins with all bits at 0; this is called the *zero count*. For each successive count, the units (2^0) position *toggles*; that is, it changes from one binary value to the other. Each time the units bit changes from a 1 to a 0, the twos (2^1) position will toggle (change states). Each time the twos position changes from 1 to 0, the fours (2^2) position will toggle (change states). Likewise, each time the fours position goes from 1 to 0, the eights (2^3) position toggles. This same process would be continued for the higher-order bit positions if the binary number had more than four bits.

The binary counting sequence has an important characteristic, as shown in Figure 1-6. The units bit (LSB) changes either from 0 to 1 or 1 to 0 with *each* count. The second bit (twos position) stays at 0 for two counts, then at 1 for two counts, then at 0 for two counts, and so on. The third bit (fours position) stays at 0 for four counts, then at 1 for four counts, and so on. The fourth bit (eights position) stays at 0 for eight counts, then at 1 for eight counts. If we wanted to

FIGURE 1-6 Binary counting sequence.



Weights → $2^3 = 8$	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$	Decimal equivalent
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9
1	0	1	0	10
1	0	1	1	11
1	1	0	0	12
1	1	0	1	13
1	1	1	0	14
1	1	1	1	15

count further, we would add more places, and this pattern would continue with 0s and 1s alternating in groups of 2^{N-1}. For example, using a fifth binary place, the fifth bit would alternate sixteen 0s, then sixteen 1s, and so on.

As we saw for the decimal system, it is also true for the binary system that by using N bits or places, we can go through 2^N counts. For example, with two bits we can go through $2^2 = 4$ counts (00_2 through 11_2); with four bits we can go through $2^4 = 16$ counts (0000_2 through 1111_2); and so on. The last count will always be all 1s and is equal to 2^N-1 in the decimal system. For example, using four bits, the last count is $1111_2 = 2^4-1 = 15_{10}$.

EXAMPLE 1-2

What is the largest number that can be represented using eight bits?

Solution

$2^N-1 = 2^8-1 = 255_{10} = 11111111_2$.

This has been a brief introduction of the binary number system and its relation to the decimal system. We will spend much more time on these two systems and several others in the next chapter.

REVIEW QUESTIONS

1. What is the decimal equivalent of 1101011_2?
2. What is the next binary number following 10111_2 in the counting sequence?
3. What is the largest decimal value that can be represented using 12 bits?

1-4 REPRESENTING BINARY QUANTITIES

In digital systems, the information being processed is usually present in binary form. Binary quantities can be represented by any device that has only two operating states or possible conditions. For example, a switch has only two states: open or closed. We can arbitrarily let an open switch represent binary 0 and a closed switch represent binary 1. With this assignment we can now represent any binary number. Figure 1-7(a) shows a binary code number for a garage door opener. The small switches are set to form the binary number 1000101010. The door will open only if a matching pattern of bits is set in the receiver and the transmitter.

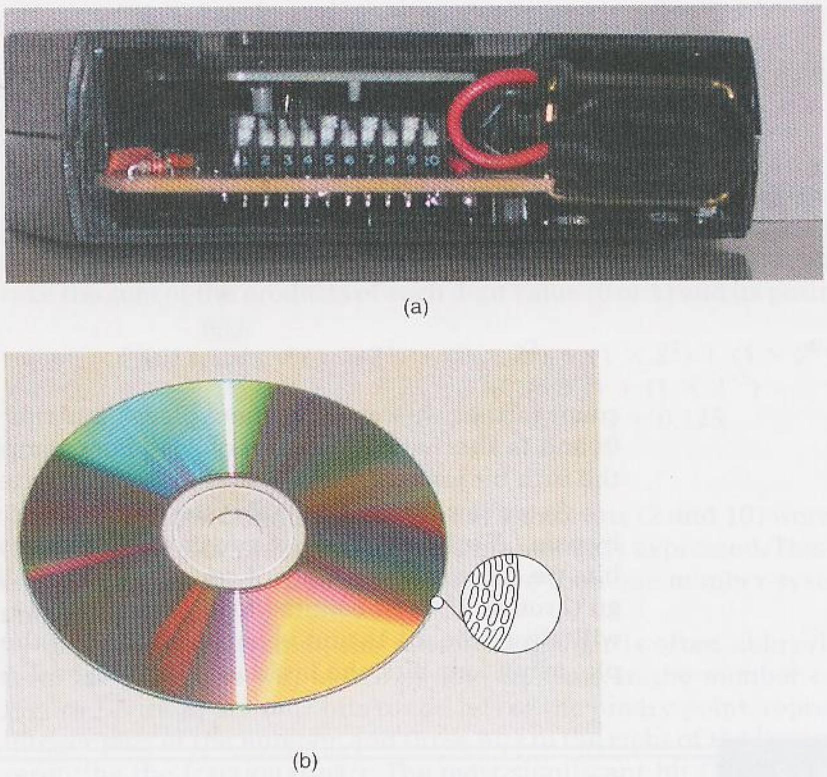


FIGURE 1-7 (a) Binary code settings for a garage door opener. (b) Digital audio on a CD.

Another example is shown in Figure 1-7(b), where binary numbers are stored on a CD. The inner surface (under a transparent plastic layer) is coated with a highly reflective aluminum layer. Holes are burned through this reflective coating to form "pits" that do not reflect light the same as the unburned areas. The areas where the pits are burned are considered "1" and the reflective areas are "0."

There are numerous other devices that have only two operating states or can be operated in two extreme conditions. Among these are: light bulb (bright or dark), diode (conducting or nonconducting), electromagnet (energized or deenergized), transistor (cut off or saturated), photocell (illuminated or dark), thermostat (open or closed), mechanical clutch (engaged or disengaged), and spot on a magnetic disk (magnetized or demagnetized).

In electronic digital systems, binary information is represented by voltages (or currents) that are present at the inputs and outputs of the various circuits. Typically, the binary 0 and 1 are represented by two nominal voltage levels. For example, zero volts (0 V) might represent binary 0, and +5 V might represent binary 1. In actuality, because of circuit variations, the 0 and 1 would be represented by voltage ranges. This is illustrated in Figure 1-8(a), where any voltage between 0 and 0.8 V represents a 0 and any voltage between 2 and 5 V represents a 1. All input and output signals will normally fall within one of these ranges, except during transitions from one level to another.

We can now see another significant difference between digital and analog systems. In digital systems, the exact value of a voltage *is not* important;

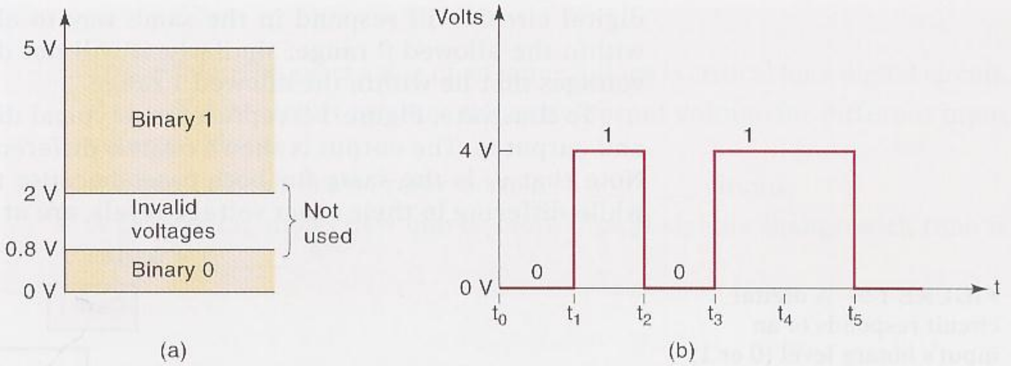


FIGURE 1-8 (a) Typical voltage assignments in digital system; (b) typical digital signal timing diagram.

for example, for the voltage assignments of Figure 1-8(a), a voltage of 3.6 V means the same as a voltage of 4.3 V. In analog systems, the exact value of a voltage *is* important. For instance, if the analog voltage is proportional to the temperature measured by a transducer, the 3.6 V would represent a different temperature than would 4.3 V. In other words, the voltage value carries significant information. This characteristic means that the design of accurate analog circuitry is generally more difficult than that of digital circuitry because of the way in which exact voltage values are affected by variations in component values, temperature, and noise (random voltage fluctuations).

Digital Signals and Timing Diagrams

Figure 1-8(b) shows a typical digital signal and how it varies over time. It is actually a graph of voltage versus time *(t)* and is called a **timing diagram**. The horizontal time scale is marked off at regular intervals beginning at t_0 and proceeding to t_1, t_2, and so on. For the example timing diagram shown here, the signal starts at 0 V (a binary 0) at time t_0 and remains there until time t_1. At t_1, the signal makes a rapid transition (jump) up to 4 V (a binary 1). At t_2, it jumps back down to 0 V. Similar transitions occur at t_3 and t_5. Note that the signal does not change at t_4 but stays at 4 V from t_3 to t_5.

The transitions on this timing diagram are drawn as vertical lines, and so they appear to be instantaneous, when in reality they are not. In many situations, however, the transition times are so short compared to the times between transitions that we can show them on the diagram as vertical lines. We will encounter situations later where it will be necessary to show the transitions more accurately on an expanded time scale.

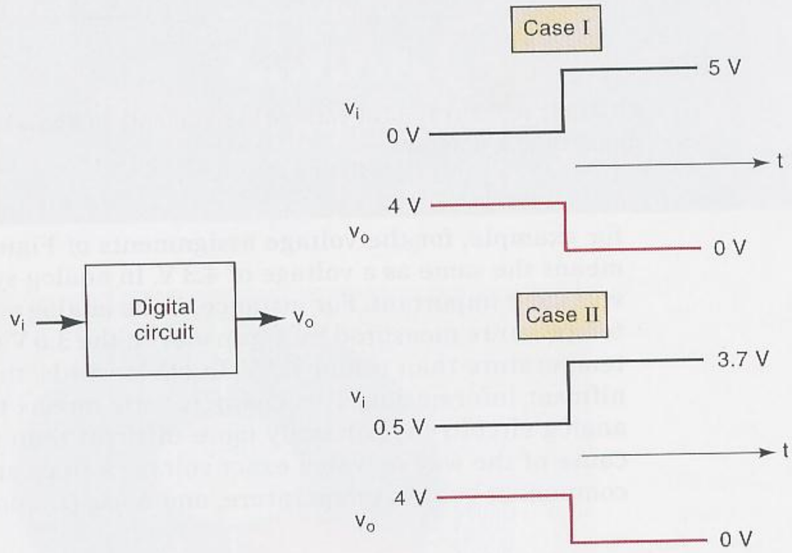
Timing diagrams are used extensively to show how digital signals change with time, and especially to show the relationship between two or more digital signals in the same circuit or system. By displaying one or more digital signals on an *oscilloscope* or *logic analyzer*, we can compare the signals to their expected timing diagrams. This is a very important part of the testing and troubleshooting procedures used in digital systems.

1-5 DIGITAL CIRCUITS/LOGIC CIRCUITS

Digital circuits are designed to produce output voltages that fall within the prescribed 0 and 1 voltage ranges such as those defined in Figure 1-8. Likewise, digital circuits are designed to respond predictably to input voltages that are within the defined 0 and 1 ranges. What this means is that a digital circuit will respond in the same way to all input voltages that fall within the allowed 0 range; similarly, it will not distinguish between input voltages that lie within the allowed 1 range.

To illustrate, Figure 1-9 represents a typical digital circuit with input v_i and output v_o. The output is shown for two different input signal waveforms. Note that v_o is the same for both cases because the two input waveforms, while differing in their exact voltage levels, are at the same binary levels.

FIGURE 1-9 A digital circuit responds to an input's binary level (0 or 1) and not to its actual voltage.



Logic Circuits

The manner in which a digital circuit responds to an input is referred to as the circuit's *logic*. Each type of digital circuit obeys a certain set of logic rules. For this reason, digital circuits are also called **logic circuits**. We will use both terms interchangeably throughout the text. In Chapter 3, we will see more clearly what is meant by a circuit's "logic."

We will be studying all the types of logic circuits that are currently used in digital systems. Initially, our attention will be focused only on the logical operation that these circuits perform—that is, the relationship between the circuit inputs and outputs. We will defer any discussion of the internal circuit operation of these logic circuits until after we have developed an understanding of their logical operation.

Digital Integrated Circuits

Almost all of the digital circuits used in modern digital systems are integrated circuits (ICs). The wide variety of available logic ICs has made it possible to construct complex digital systems that are smaller and more reliable than their discrete-component counterparts.

Several integrated-circuit fabrication technologies are used to produce digital ICs, the most common being CMOS, TTL, NMOS, and ECL. Each differs in the type of circuitry used to provide the desired logic operation. For example, TTL (transistor-transistor logic) uses the bipolar transistor as its main circuit element, while CMOS (complementary metal-oxide-semiconductor) uses the enhancement-mode MOSFET as its principal circuit element. We will learn about the various IC technologies, their characteristics, and their relative advantages and disadvantages after we master the basic logic circuit types.

REVIEW QUESTIONS

1. *True or false:* The exact value of an input voltage is critical for a digital circuit.
2. Can a digital circuit produce the same output voltage for different input voltage values?
3. A digital circuit is also referred to as a _______ circuit.
4. A graph that shows how one or more digital signals change with time is called a _______.

1-6 PARALLEL AND SERIAL TRANSMISSION

One of the most common operations that occur in any digital system is the transmission of information from one place to another. The information can be transmitted over a distance as small as a fraction of an inch on the same circuit board, or over a distance of many miles when an operator at a computer terminal is communicating with a computer in another city. The information that is transmitted is in binary form and is generally represented as voltages at the outputs of a sending circuit that are connected to the inputs of a receiving circuit. Figure 1-10 illustrates the two basic methods for digital information transmission: **parallel** and **serial**.

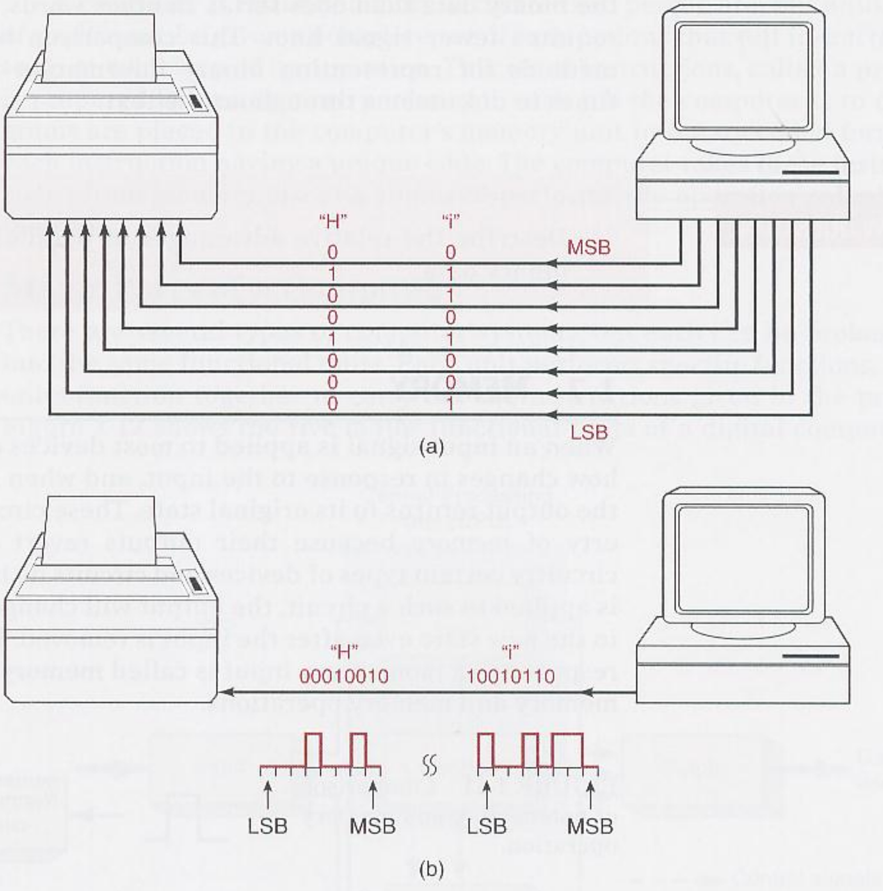


FIGURE 1-10 (a) Parallel transmission uses one connecting line per bit, and all bits are transmitted simultaneously; (b) serial transmission uses only one signal line, and the individual bits are transmitted serially (one at a time).

Figure 1-10(a) demonstrates parallel transmission of data from a computer to a printer using the parallel printer port (LPT1) of the computer. In this scenario, assume we are trying to print the word "Hi" on the printer. The binary code for "H" is 01001000 and the binary code for "i" is 01101001. Each character (the "H" and the "i") are made up of eight bits. Using parallel transmission, all eight bits are sent simultaneously over eight wires. The "H" is sent first, followed by the "i."

Figure 1-10(b) demonstrates serial transmission such as is employed when using a serial COM port on your computer to send data to a modem, or when using a USB (Universal Serial Bus) port to send data to a printer. Although the details of the data format and speed of transmission are quite different between a COM port and a USB port, the actual data are sent in the same way: one bit at a time over a single wire. The bits are shown in the diagram as though they were actually moving down the wire in the order shown. The least significant bit of "H" is sent first and the most significant bit of "i" is sent last. Of course, in reality, only one bit can be on the wire at any point in time and time is usually drawn on a graph starting at the left and advancing to the right. This produces a graph of logic bits versus time of the serial transmission called a timing diagram. Notice that in this presentation, the least significant bit is shown on the left because it was sent first.

The principal trade-off between parallel and serial representations is one of speed versus circuit simplicity. The transmission of binary data from one part of a digital system to another can be done more quickly using parallel representation because all the bits are transmitted simultaneously, while serial representation transmits one bit at a time. On the other hand, parallel requires more signal lines connected between the sender and the receiver of the binary data than does serial. In other words, parallel is faster, and serial requires fewer signal lines. This comparison between parallel and serial methods for representing binary information will be encountered many times in discussions throughout the text.

REVIEW QUESTION

1. Describe the relative advantages of parallel and serial transmission of binary data.

1-7 MEMORY

When an input signal is applied to most devices or circuits, the output somehow changes in response to the input, and when the input signal is removed, the output returns to its original state. These circuits do not exhibit the property of *memory* because their outputs revert back to normal. In digital circuitry certain types of devices and circuits do have memory. When an input is applied to such a circuit, the output will change its state, but it will remain in the new state even after the input is removed. This property of retaining its response to a momentary input is called **memory**. Figure 1-11 illustrates nonmemory and memory operations.

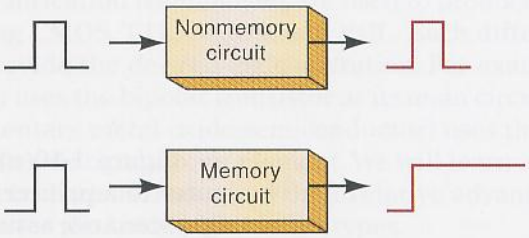


FIGURE 1-11 Comparison of nonmemory and memory operation.

Memory devices and circuits play an important role in digital systems because they provide a means for storing binary numbers either temporarily or permanently, with the ability to change the stored information at any time. As we shall see, the various memory elements include magnetic and optical types and those that utilize electronic latching circuits (called *latches* and *flip-flops*).

1-8 DIGITAL COMPUTERS

Digital techniques have found their way into innumerable areas of technology, but the area of automatic **digital computers** is by far the most notable and most extensive. Although digital computers affect some part of all of our lives, it is doubtful that many of us know exactly what a computer does. In simplest terms, *a computer is a system of hardware that performs arithmetic operations, manipulates data (usually in binary form), and makes decisions.*

For the most part, human beings can do whatever computers can do, but computers can do it with much greater speed and accuracy, in spite of the fact that computers perform all their calculations and operations one step at a time. For example, a human being can take a list of 10 numbers and find their sum all in one operation by listing the numbers one over the other and adding them column by column. A computer, on the other hand, can add numbers only two at a time, so that adding this same list of numbers will take nine actual addition steps. Of course, the fact that the computer requires only a few nanoseconds per step makes up for this apparent inefficiency.

A computer is faster and more accurate than people are, but unlike most of us, it must be given a complete set of instructions that tell it *exactly* what to do at each step of its operation. This set of instructions, called a **program**, is prepared by one or more persons for each job the computer is to do. Programs are placed in the computer's memory unit in binary-coded form, with each instruction having a unique code. The computer takes these instruction codes from memory *one at a time* and performs the operation called for by the code.

Major Parts of a Computer

There are several types of computer systems, but each can be broken down into the same functional units. Each unit performs specific functions, and all units function together to carry out the instructions given in the program. Figure 1-12 shows the five major functional parts of a digital computer and

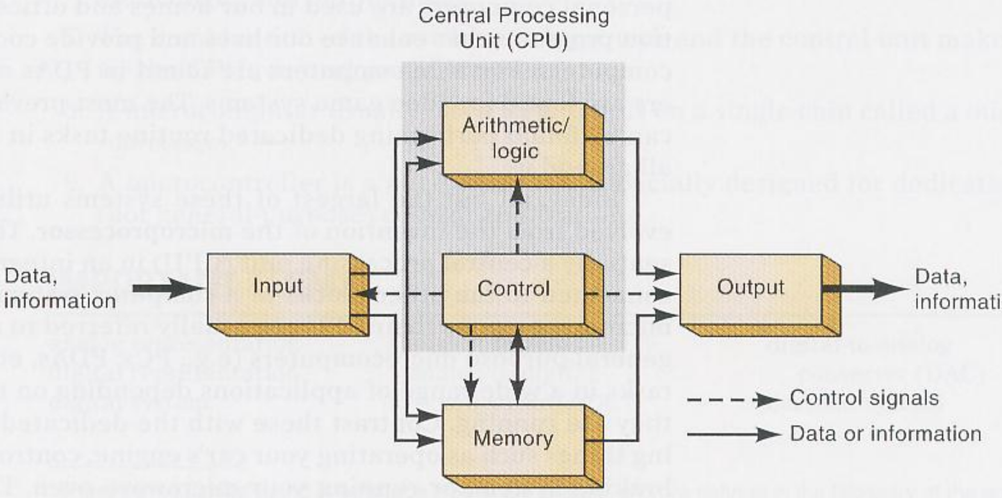


FIGURE 1-12 Functional diagram of a digital computer.

their interaction. The solid lines with arrows represent the flow of data and information. The dashed lines with arrows represent the flow of timing and control signals.

The major functions of each unit are:

1. **Input unit**. Through this unit, a complete set of instructions and data is fed into the computer system and into the memory unit, to be stored until needed. The information typically enters the input unit from a keyboard or a disk.
2. **Memory unit**. The memory stores the instructions and data received from the input unit. It stores the results of arithmetic operations received from the arithmetic unit. It also supplies information to the output unit.
3. **Control unit**. This unit takes instructions from the memory unit one at a time and interprets them. It then sends appropriate signals to all the other units to cause the specific instruction to be executed.
4. **Arithmetic/logic unit**. All arithmetic calculations and logical decisions are performed in this unit, which can then send results to the memory unit to be stored.
5. **Output unit**. This unit takes data from the memory unit and prints out, displays, or otherwise presents the information to the operator (or process, in the case of a process control computer).

Central Processing Unit (CPU)

As the diagram in Figure 1-12 shows, the control and arithmetic/logic units are often considered as one unit, called the **central processing unit (CPU)**. The CPU contains all of the circuitry for fetching and interpreting instructions and for controlling and performing the various operations called for by the instructions.

TYPES OF COMPUTERS All computers are made up of the basic units described above, but they can differ as to physical size, operating speed, memory capacity, and computational power, as well as other characteristics. Computer systems are configured in many and various ways today, with many common characteristics and distinguishing differences. Large computer systems that are permanently installed in multiple cabinets are used by corporations and universities for information technology support. Desktop personal computers are used in our homes and offices to run useful application programs that enhance our lives and provide communication with other computers. Portable computers are found in PDAs and specialized computers are found in video game systems. The most prevalent form of computers can be found performing dedicated routine tasks in appliances and systems all around us.

Today, all but the largest of these systems utilize technology that has evolved from the invention of the **microprocessor**. The microprocessor is essentially a central processing unit (CPU) in an integrated circuit that can be connected to the other blocks of a computer system. Computers that use a microprocessor as their CPU are usually referred to as **microcomputers**. The general-purpose microcomputers (e.g., PCs, PDAs, etc.) perform a variety of tasks in a wide range of applications depending on the software (programs) they are running. Contrast these with the dedicated computers that are doing things such as operating your car's engine, controlling your car's antilock braking system, or running your microwave oven. These computers cannot be programmed by the user, but simply perform their intended control task: they are referred to as **microcontrollers**. Since these microcontrollers are an integral part of a bigger system and serve a dedicated purpose, they also are called *embedded controllers*. Microcontrollers generally have all the elements of a complete computer (CPU, memory, and input/output ports), all contained on a single integrated circuit. You can find them embedded in your kitchen appliances, entertainment equipment, photocopiers, automatic teller machines, automated manufacturing equipment, medical instrumentation, and much, much more.

So you see, even people who don't own a PC or use one at work or school are using microcomputers every day because so many modern consumer electronic devices, appliances, office equipment, and much more are built around embedded microcontrollers. If you work, play, or go to school in this digital age, there's no escaping it: you'll use a microcomputer somewhere.

REVIEW QUESTIONS

1. Explain how a digital circuit that has memory differs from one that does not.
2. Name the five major functional units of a computer.
3. Which two units make up the CPU?
4. An IC chip that contains a CPU is called a _____.

SUMMARY

1. The two basic ways of representing the numerical value of physical quantities are analog (continuous) and digital (discrete).
2. Most quantities in the real world are analog, but digital techniques are generally superior to analog techniques, and most of the predicted advances will be in the digital realm.
3. The binary number system (0 and 1) is the basic system used in digital technology.
4. Digital or logic circuits operate on voltages that fall in prescribed ranges that represent either a binary 0 or a binary 1.
5. The two basic ways to transfer digital information are parallel—all bits simultaneously—and serial—one bit at a time.
6. The main parts of all computers are the input, control, memory, arithmetic/logic, and output units.
7. The combination of the arithmetic/logic unit and the control unit makes up the CPU (central processing unit).
8. A microcomputer usually has a CPU that is on a single chip called a *microprocessor.*
9. A microcontroller is a microcomputer especially designed for dedicated (not general-purpose) control applications.

IMPORTANT TERMS*

analog representation
digital representation
digital system
analog system
analog-to-digital converter (ADC)
digital-to-analog converter (DAC)
decimal system

---

*These terms can be found in **boldface** type in the chapter and are defined in the Glossary at the end of the book. This applies to all chapters.

- binary system
- bit
- timing diagram
- digital circuits/logic circuits
- parallel transmission
- serial transmission
- memory
- digital computer
- program
- input unit
- memory unit
- control unit
- arithmetic/logic unit
- output unit
- central processing unit (CPU)
- microprocessor
- microcomputer
- microcontroller

PROBLEMS

SECTION 1-2

1-1.* Which of the following are analog quantities, and which are digital?

(a) Number of atoms in a sample of material

(b) Altitude of an aircraft

(c) Pressure in a bicycle tire

(d) Current through a speaker

(e) Timer setting on a microwave oven

1-2. Which of the following are analog quantities, and which are digital?

(a) Width of a piece of lumber

(b) The amount of time before the oven buzzer goes off

(c) The time of day displayed on a quartz watch

(d) Altitude above sea level measured on a staircase

(e) Altitude above sea level measured on a ramp

SECTION 1-3

1-3.* Convert the following binary numbers to their equivalent decimal values.

(a) 11001_2

(b) 1001.1001_2

(c) 10011011001.10110_2

1-4. Convert the following binary numbers to decimal.

(a) 10011_2

(b) 1100.0101

(c) 10011100100.10010

1-5.* Using three bits, show the binary counting sequence from 000 to 111.

1-6. Using six bits, show the binary counting sequence from 000000 to 111111.

1-7.* What is the maximum number that we can count up to using 10 bits?

1-8. What is the maximum number that we can count up to using 14 bits?

1-9.* How many bits are needed to count up to a maximum of 511?

1-10. How many bits are needed to count up to a maximum of 63?

SECTION 1-4

1-11.* Draw the timing diagram for a digital signal that continuously alternates between 0.2 V (binary 0) for 2 ms and 4.4 V (binary 1) for 4 ms.

---

*Answers to problems marked with an asterisk can be found in the back of the text.

1-12. Draw the timing diagram for a signal that alternates between 0.3 V (binary 0) for 5 ms and 3.9 V (binary 1) for 2 ms.

SECTION 1-6

1-13.*Suppose that the decimal integer values from 0 to 15 are to be transmitted in binary.

(a) How many lines will be needed if parallel representation is used?

(b) How many will be needed if serial representation is used?

SECTIONS 1-7 AND 1-8

1-14. How is a microprocessor different from a microcomputer?

1-15. How is a microcontroller different from a microcomputer?

ANSWERS TO SECTION REVIEW QUESTIONS

SECTION 1-1

1. Analog quantities can take on *any* value over a continuous range; digital quantities can take on only *discrete* values.

SECTION 1-2

1. Easier to design; easier to store information; greater accuracy and precision; programmability; less affected by noise; higher degree of integration
2. Real-world physical quantities are analog. Digital processing takes time.

SECTION 1-3

1. 107_{10} 2. 11000_2 3. 4095_{10}

SECTION 1-5

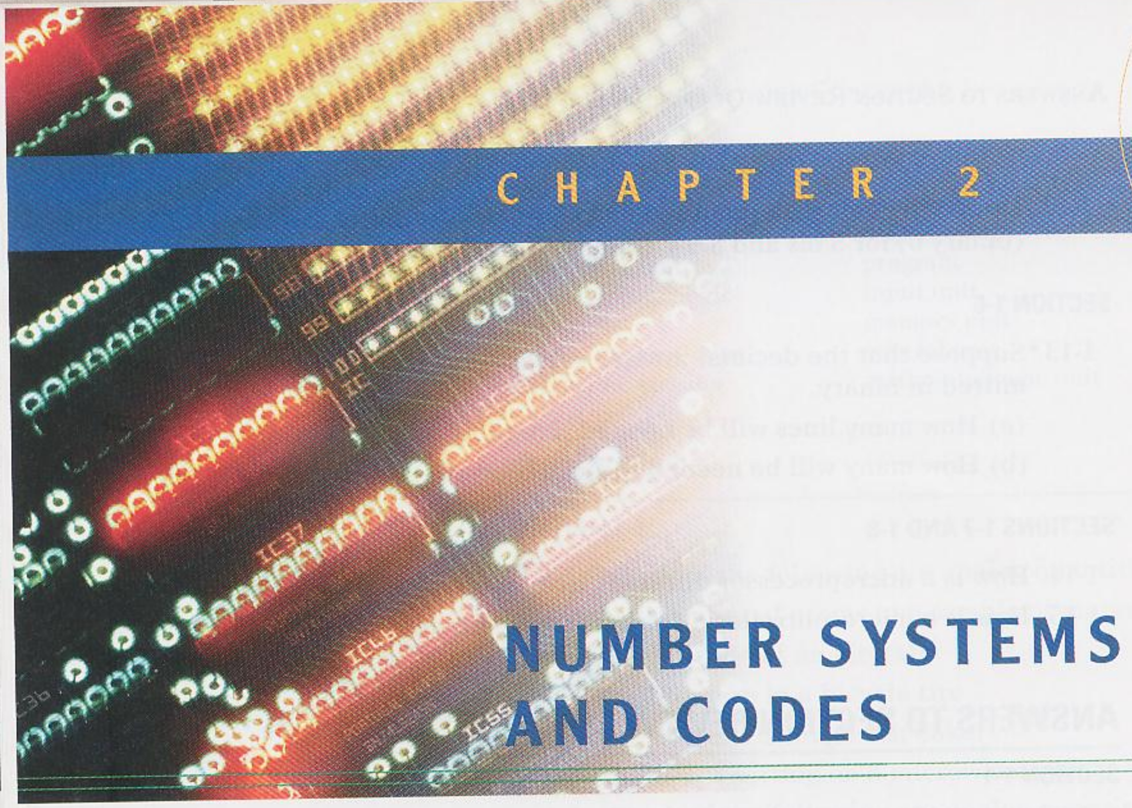
1. False 2. Yes, provided that the two input voltages are within the same logic level range 3. Logic 4. Timing diagram

SECTION 1-6

1. Parallel is faster; serial requires only one signal line.

SECTION 1-8

1. One that has memory will have its output changed and *remain* changed in response to a momentary change in the input signal. 2. Input, output, memory, arithmetic/logic, control 3. Control and arithmetic/logic 4. Microprocessor



CHAPTER 2

NUMBER SYSTEMS AND CODES

OUTLINE

2-1 Binary-to-Decimal Conversions
2-2 Decimal-to-Binary Conversions
2-3 Hexadecimal Number System
2-4 BCD Code
2-5 The Gray Code
2-6 Putting It All Together
2-7 The Byte, Nibble, and Word
2-8 Alphanumeric Codes
2-9 Parity Method for Error Detection
2-10 Applications

OBJECTIVES

Upon completion of this chapter, you will be able to:

- Convert a number from one number system (decimal, binary, hexadecimal) to its equivalent in one of the other number systems.
- Cite the advantages of the hexadecimal number system.
- Count in hexadecimal.
- Represent decimal numbers using the BCD code; cite the pros and cons of using BCD.
- Understand the difference between BCD and straight binary.
- Understand the purpose of alphanumeric codes such as the ASCII code.
- Explain the parity method for error detection.
- Determine the parity bit to be attached to a digital data string.

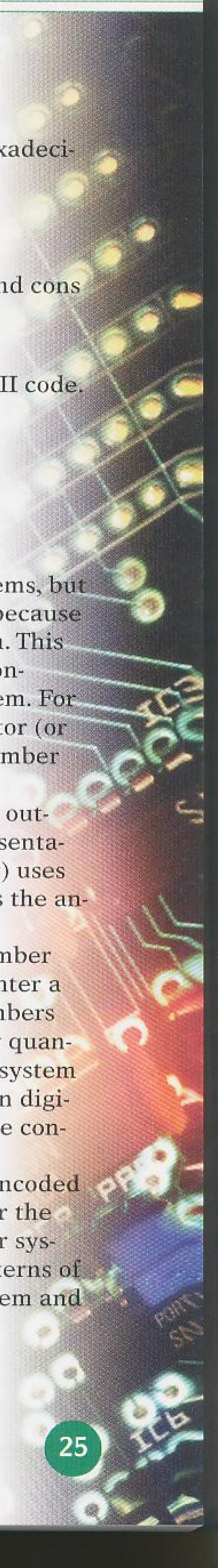
INTRODUCTION

The binary number system is the most important one in digital systems, but several others are also important. The decimal system is important because it is universally used to represent quantities outside a digital system. This means that there will be situations where decimal values must be converted to binary values before they are entered into the digital system. For example, when you punch a decimal number into your hand calculator (or computer), the circuitry inside the machine converts the decimal number to a binary value.

Likewise, there will be situations where the binary values at the outputs of a digital system must be converted to decimal values for presentation to the outside world. For example, your calculator (or computer) uses binary numbers to calculate answers to a problem and then converts the answers to decimal digits before displaying them.

As you will see, it is not easy to simply look at a large binary number and convert it to its equivalent decimal value. It is very tedious to enter a long sequence of 1s and 0s on a keypad, or to write large binary numbers on a piece of paper. It is especially difficult to try to convey a binary quantity while speaking to someone. The hexadecimal (base-16) number system has become a very standard way of communicating numeric values in digital systems. The great advantage is that hexadecimal numbers can be converted easily to and from binary.

Other methods of representing decimal quantities with binary-encoded digits have been devised that are not truly number systems but offer the ease of conversion between the binary code and the decimal number system. This is referred to as binary-coded decimal. Quantities and patterns of bits might be represented by any of these methods in any given system and



throughout the written material that supports the system, so it is very important that you are able to interpret values in any system and convert between any of these numeric representations. Other codes that use 1s and 0s to represent things such as alphanumeric characters will be covered because they are so common in digital systems.

2-1 BINARY-TO-DECIMAL CONVERSIONS

As explained in Chapter 1, the binary number system is a positional system where each binary digit (bit) carries a certain weight based on its position relative to the LSB. Any binary number can be converted to its decimal equivalent simply by summing together the weights of the various positions in the binary number that contain a 1. To illustrate, let's change 11011_2 to its decimal equivalent.

$$\begin{array}{l} 1 \quad\; 1 \quad\; 0 \quad\; 1 \quad\; 1_2 \\ 2^4 + 2^3 + 0 + 2^1 + 2^0 = 16 + 8 + 2 + 1 \\ \qquad\qquad\qquad\qquad\quad = 27_{10} \end{array}$$

Let's try another example with a greater number of bits:

$$\begin{array}{l} 1 \quad\;\; 0 \quad\;\; 1 \quad\;\; 1 \quad\;\; 0 \quad\;\; 1 \quad\;\; 0 \quad\;\; 1_2 = \\ 2^7 + 0 + 2^5 + 2^4 + 0 + 2^2 + 0 + 2^0 = 181_{10} \end{array}$$

Note that the procedure is to find the weights (i.e., powers of 2) for each bit position that contains a 1, and then to add them up. Also note that the MSB has a weight of 2^7 even though it is the eighth bit; this is because the LSB is the first bit and has a weight of 2^0.

REVIEW QUESTIONS

1. Convert 100011011011_2 to its decimal equivalent.
2. What is the weight of the MSB of a 16-bit number?

2-2 DECIMAL-TO-BINARY CONVERSIONS

There are two ways to convert a decimal *whole* number to its equivalent binary-system representation. The first method is the reverse of the process described in Section 2-1. The decimal number is simply expressed as a sum of powers of 2, and then 1s and 0s are written in the appropriate bit positions. To illustrate:

$$\begin{array}{l} 45_{10} = 32 + 8 + 4 + 1 = 2^5 + 0 + 2^3 + 2^2 + 0 + 2^0 \\ \qquad\qquad\qquad\qquad\quad\;\; = 1 \quad\; 0 \quad\; 1 \quad\;\; 1 \quad\; 0 \quad\; 1_2 \end{array}$$

Note that a 0 is placed in the 2^1 and 2^4 positions, since all positions must be accounted for. Another example is the following:

$$\begin{array}{l} 76_{10} = 64 + 8 + 4 = 2^6 + 0 + 0 + 2^3 + 2^2 + 0 + 0 \\ \qquad\qquad\qquad\quad\;\; = 1 \quad\; 0 \quad\; 0 \quad\; 1 \quad\;\; 1 \quad\; 0 \quad\; 0_2 \end{array}$$

Repeated Division

Another method for converting decimal integers uses repeated division by 2. The conversion, illustrated below for 25_{10}, requires repeatedly dividing the decimal number by 2 and writing down the remainder after each division until a quotient of 0 is obtained. Note that the binary result is obtained by writing the first remainder as the LSB and the last remainder as the MSB. This process, diagrammed in the flowchart of Figure 2-1, can also be used to convert from decimal to any other number system, as we shall see.

$$\frac{25}{2} = \boxed{12} + \text{remainder of } 1 \quad \text{LSB}$$

$$\frac{\boxed{12}}{2} = 6 + \text{remainder of } 0$$

$$\frac{6}{2} = 3 + \text{remainder of } 0$$

$$\frac{3}{2} = 1 + \text{remainder of } 1$$

$$\frac{1}{2} = 0 + \text{remainder of } 1 \quad \text{MSB}$$

$$25_{10} = \boxed{1\ 1\ 0\ 0\ 1_2}$$

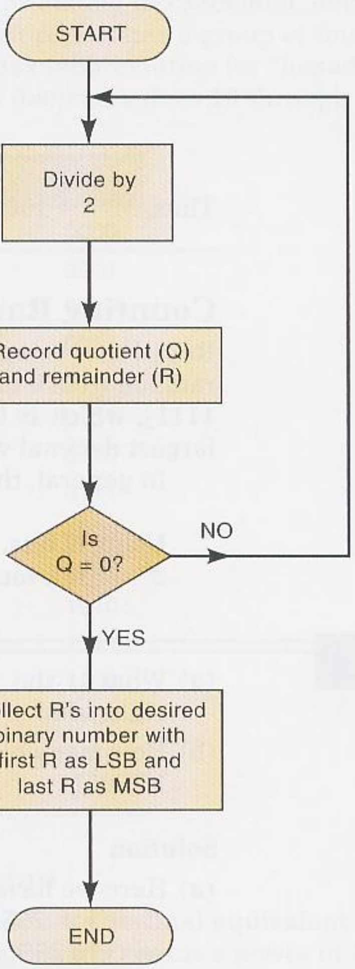

START
Divide by 2
Record quotient (Q) and remainder (R)
Is Q = 0?
NO
YES
Collect R's into desired binary number with first R as LSB and last R as MSB
END

FIGURE 2-1 Flowchart for repeated-division method of decimal-to-binary conversion of integers. The same process can be used to convert a decimal integer to any other number system.

CALCULATOR HINT:

If you use a calculator to perform the divisions by 2, you can tell whether the remainder is 0 or 1 by whether or not the result has a fractional part. For instance, 25/2 would produce 12.5. Since there is a fractional part (the .5), the remainder is a 1. If there were no fractional part, such as 12/2 = 6, then the remainder would be 0. The following example illustrates this.

EXAMPLE 2-1

Convert 37_{10} to binary. Try to do it on your own before you look at the solution.

Solution

$$\begin{array}{lll} \frac{37}{2} = 18.5 \longrightarrow & \text{remainder of 1 (LSB)} \\ \frac{18}{2} = 9.0 \longrightarrow & & 0 \\ \frac{9}{2} = 4.5 \longrightarrow & & 1 \\ \frac{4}{2} = 2.0 \longrightarrow & & 0 \\ \frac{2}{2} = 1.0 \longrightarrow & & 0 \\ \frac{1}{2} = 0.5 \longrightarrow & & 1 \text{ (MSB)} \end{array}$$

Thus, $37^{10} = \mathbf{100101_2}$.

Counting Range

Recall that using N bits, we can count through 2^N different decimal numbers ranging from 0 to $2^N - 1$. For example, for $N = 4$, we can count from 0000_2 to 1111_2, which is 0_{10} to 15_{10}, for a total of 16 different numbers. Here, the largest decimal value is $2^4 - 1 = 15$, and there are 2^4 different numbers.

In general, then, we can state:

Using N bits, we can represent decimal numbers ranging from 0 to $2^N - 1$, a total of 2^N different numbers.

EXAMPLE 2-2

(a) What is the total range of decimal values that can be represented in eight bits?

(b) How many bits are needed to represent decimal values ranging from 0 to 12,500?

Solution

(a) Here we have $N = 8$. Thus, we can represent decimal numbers from 0 to $2^8 - 1 = \mathbf{255}$. We can verify this by checking to see that 11111111_2 converts to 255_{10}.

(b) With 13 bits, we can count from decimal 0 to $2^{13} - 1 = 8191$. With 14 bits, we can count from 0 to $2^{14} - 1 = 16{,}383$. Clearly, 13 bits aren't enough, but 14 bits will get us up beyond 12,500. Thus, the required number of bits is **14**.

REVIEW QUESTIONS

1. Convert 83_{10} to binary using both methods.
2. Convert 729_{10} to binary using both methods. Check your answer by converting back to decimal.
3. How many bits are required to count up to decimal 1 million?

2-3 HEXADECIMAL NUMBER SYSTEM

The **hexadecimal number system** uses base 16. Thus, it has 16 possible digit symbols. It uses the digits 0 through 9 plus the letters A, B, C, D, E, and F as the 16 digit symbols. The digit positions are weighted as powers of 16 as shown below, rather than as powers of 10 as in the decimal system.

16^4	16^3	16^2	16^1	16^0	16^{-1}	16^{-2}	16^{-3}	16^{-4}

Hexadecimal point

Table 2-1 shows the relationships among hexadecimal, decimal, and binary. Note that each hexadecimal digit represents a group of four binary digits. It is important to remember that hex (abbreviation for "hexadecimal") digits A through F are equivalent to the decimal values 10 through 15.

TABLE 2-1

Hexadecimal	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Hex-to-Decimal Conversion

A hex number can be converted to its decimal equivalent by using the fact that each hex digit position has a weight that is a power of 16. The LSD has a weight of $16^0 = 1$; the next higher digit position has a weight of $16^1 = 16$; the next has a weight of $16^2 = 256$; and so on. The conversion process is demonstrated in the examples below.

CALCULATOR HINT:

You can use the y^x calculator function to evaluate the powers of 16.

$$\begin{aligned} 356_{16} &= 3 \times 16^2 + 5 \times 16^1 + 6 \times 16^0 \\ &= 768 + 80 + 6 \\ &= 854_{10} \end{aligned}$$

$$\begin{aligned} 2AF_{16} &= 2 \times 16^2 + 10 \times 16^1 + 15 \times 16^0 \\ &= 512 + 160 + 15 \\ &= 687_{10} \end{aligned}$$

Note that in the second example, the value 10 was substituted for A and the value 15 for F in the conversion to decimal.

For practice, verify that $1BC2_{16}$ is equal to 7106_{10}.

Decimal-to-Hex Conversion

Recall that we did decimal-to-binary conversion using repeated division by 2. Likewise, decimal-to-hex conversion can be done using repeated division by 16 (Figure 2-1). The following example contains two illustrations of this conversion.

EXAMPLE 2-3

(a) Convert 423_{10} to hex.

Solution

$$\frac{423}{16} = \boxed{26} + \text{remainder of } 7$$

$$\frac{\boxed{26}}{16} = 1 + \text{remainder of } 10$$

$$\frac{1}{16} = 0 + \text{remainder of } 1$$

$$423_{10} = \boxed{1A7_{16}}$$

(b) Convert 214_{10} to hex.

Solution

$$\frac{214}{16} = 13 + \text{remainder of } 6$$

$$\frac{13}{16} = 0 + \text{remainder of } 13$$

$$214_{10} = \boxed{D6_{16}}$$

Again note that the remainders of the division processes form the digits of the hex number. Also note that any remainders that are greater than 9 are represented by the letters A through F.

CALCULATOR HINT:

If a calculator is used to perform the divisions in the conversion process, the results will include a decimal fraction instead of a remainder. The remainder can be obtained by multiplying the fraction by 16. To illustrate, in Example 2-3(b), the calculator would have produced

$$\frac{214}{16} = 13.375$$

The remainder becomes $(0.375) \times 16 = 6$.

Hex-to-Binary Conversion

The hexadecimal number system is used primarily as a "shorthand" method for representing binary numbers. It is a relatively simple matter to convert a hex number to binary. *Each* hex digit is converted to its four-bit binary equivalent (Table 2-1). This is illustrated below for $9F2_{16}$.

$$\begin{array}{llll} 9F2_{16} = & 9 & F & 2 \\ & \downarrow & \downarrow & \downarrow \\ & = 1\ 0\ 0\ 1 & 1\ 1\ 1\ 1 & 0\ 0\ 1\ 0 \\ & = 100111110010_2 & & \end{array}$$

For practice, verify that $BA6_{16} = 101110100110_2$.

Binary-to-Hex Conversion

Conversion from binary to hex is just the reverse of the process above. The binary number is grouped into groups of *four* bits, and each group is converted to its equivalent hex digit. Zeros (shown shaded) are added, as needed, to complete a four-bit group.

$$1\,1\,1\,0\,1\,0\,0\,1\,1\,0_2 = \underbrace{0\,0\,1\,1}_{3}\ \underbrace{1\,0\,1\,0}_{A}\ \underbrace{0\,1\,1\,0}_{6}$$
$$= 3A6_{16}$$

To perform these conversions between hex and binary, it is necessary to know the four-bit binary numbers (0000 through 1111) and their equivalent hex digits. Once these are mastered, the conversions can be performed quickly without the need for any calculations. This is why hex is so useful in representing large binary numbers.

For practice, verify that $101011111_2 = 15F_{16}$.

Counting in Hexadecimal

When counting in hex, each digit position can be incremented (increased by 1) from 0 to F. Once a digit position reaches the value F, it is reset to 0, and the next digit position is incremented. This is illustrated in the following hex counting sequences:

(a) 38, 39, 3A, 3B, 3C, 3D, 3E, 3F, 40, 41, 42

(b) 6F8, 6F9, 6FA, 6FB, 6FC, 6FD, 6FE, 6FF, 700

Note that when there is a 9 in a digit position, it becomes an A when it is incremented.

With N hex digit positions, we can count from decimal 0 to $16^N - 1$, for a total of 16^N different values. For example, with three hex digits, we can count from 000_{16} to FFF_{16}, which is 0_{10} to 4095_{10}, for a total of $4096 = 16^3$ different values.

Usefulness of Hex

Hex is often used in a digital system as sort of a "shorthand" way to represent strings of bits. In computer work, strings as long as 64 bits are not uncommon. These binary strings do not always represent a numerical value, but—as you will find out—can be some type of code that conveys nonnumerical information. When dealing with a large number of bits, it is more convenient and less error-prone to write the binary numbers in hex and, as we have seen, it is relatively easy to convert back and forth between binary and hex. To illustrate the advantage of hex representation of a binary string, suppose you had in front of you a printout of the contents of 50 memory locations, each of which was a 16-bit number, and you were checking it against a list. Would you rather check 50 numbers like this one: 0110111001100111, or 50 numbers like this one: 6E67? And which one would you be more apt to read incorrectly? It is important to keep in mind, though, that digital circuits all work in binary. Hex is simply used as a convenience for the humans involved. You should memorize the 4-bit binary pattern for each hexadecimal digit. Only then will you realize the usefulness of this tool in digital systems.

EXAMPLE 2-4

Convert decimal 378 to a 16-bit binary number by first converting to hexadecimal.

Solution

$$\frac{378}{16} = 23 + \text{remainder of } 10_{10} = A_{16}$$

$$\frac{23}{16} = 1 + \text{remainder of } 7$$

$$\frac{1}{16} = 0 + \text{remainder of } 1$$

Thus, $378_{10} = 17A_{16}$. This hex value can be converted easily to binary 000101111010. Finally, we can express 378_{10} as a 16-bit number by adding four leading 0s:

$$378_{10} = 0000 \quad 0001 \quad 0111 \quad 1010_2$$

EXAMPLE 2-5

Convert $B2F_{16}$ to decimal.

Solution

$$\begin{aligned} B2F_{16} &= B \times 16^2 + 2 \times 16^1 + F \times 16^0 \\ &= 11 \times 256 + 2 \times 16 + 15 \\ &= 2863_{10} \end{aligned}$$

Summary of Conversions

Right now, your head is probably spinning as you try to keep straight all of these different conversions from one number system to another. You probably realize that many of these conversions can be done *automatically* on your calculator just by pressing a key, but it is important for you to master these conversions so that you understand the process. Besides, what happens if your calculator battery dies at a crucial time and you have no handy replacement? The following summary should help you, but nothing beats practice, practice, practice!

1. When converting from binary [or hex] to decimal, use the method of taking the weighted sum of each digit position.
2. When converting from decimal to binary [or hex], use the method of repeatedly dividing by 2 [or 16] and collecting remainders (Figure 2-1).
3. When converting from binary to hex, group the bits in groups of four, and convert each group into the correct hex digit.
4. When converting from hex to binary, convert each digit into its four-bit equivalent.

REVIEW QUESTIONS

1. Convert $24CE_{16}$ to decimal.
2. Convert 3117_{10} to hex, then from hex to binary.
3. Convert 1001011110110101_2 to hex.
4. Write the next four numbers in this hex counting sequence: E9A, E9B, E9C, E9D, ____, ____, ____, ____.
5. Convert 3527 to binary_{16}.
6. What range of decimal values can be represented by a four-digit hex number?

2-4 BCD CODE

When numbers, letters, or words are represented by a special group of symbols, we say that they are being encoded, and the group of symbols is called a *code*. Probably one of the most familiar codes is the Morse code, where a series of dots and dashes represents letters of the alphabet.

We have seen that any decimal number can be represented by an equivalent binary number. The group of 0s and 1s in the binary number can be thought of as a code representing the decimal number. When a decimal number is represented by its equivalent binary number, we call it **straight binary coding**.

Digital systems all use some form of binary numbers for their internal operation, but the external world is decimal in nature. This means that conversions between the decimal and binary systems are being performed often. We have seen that the conversions between decimal and binary can become long and complicated for large numbers. For this reason, a means of encoding decimal numbers that combines some features of both the decimal and the binary systems is used in certain situations.

Binary-Coded-Decimal Code

If *each* digit of a decimal number is represented by its binary equivalent, the result is a code called **binary-coded-decimal** (hereafter abbreviated BCD). Since a decimal digit can be as large as 9, four bits are required to code each digit (the binary code for 9 is 1001).

To illustrate the BCD code, take a decimal number such as 874. Each *digit* is changed to its binary equivalent as follows:

$$\begin{array}{cccl} 8 & 7 & 4 & \text{(decimal)} \\ \downarrow & \downarrow & \downarrow & \\ 1000 & 0111 & 0100 & \text{(BCD)} \end{array}$$

As another example, let us change 943 to its BCD-code representation:

$$\begin{array}{cccl} 9 & 4 & 3 & \text{(decimal)} \\ \downarrow & \downarrow & \downarrow & \\ 1001 & 0100 & 0011 & \text{(BCD)} \end{array}$$

Once again, each decimal digit is changed to its straight binary equivalent. Note that four bits are *always* used for each digit.

The BCD code, then, represents each digit of the decimal number by a four-bit binary number. Clearly only the four-bit binary numbers from 0000 through 1001 are used. The BCD code does not use the numbers 1010, 1011, 1100, 1101, 1110, and 1111. In other words, only 10 of the 16 possible four-bit binary code groups are used. If any of the "forbidden" four-bit numbers ever occurs in a machine using the BCD code, it is usually an indication that an error has occurred.

EXAMPLE 2-6

Convert 0110100000111001 (BCD) to its decimal equivalent.

Solution

Divide the BCD number into four-bit groups and convert each to decimal.

$$\underbrace{0110}_{6}\,\underbrace{1000}_{8}\,\underbrace{0011}_{3}\,\underbrace{1001}_{9}$$

EXAMPLE 2-7

Convert the BCD number 011111000001 to its decimal equivalent.

Solution

$$\underbrace{0111}_{7}\;\underset{\downarrow}{1100}\;\underbrace{0001}_{1}$$

The forbidden code group indicates an error in the BCD number

Comparison of BCD and Binary

It is important to realize that BCD is not another number system like binary, decimal, and hexadecimal. In fact, it is the decimal system with each digit encoded in its binary equivalent. It is also important to understand that a BCD number is *not* the same as a straight binary number. A straight binary number takes the *complete* decimal number and represents it in binary; the BCD code converts *each* decimal *digit* to binary individually. To illustrate, take the number 137 and compare its straight binary and BCD codes:

$$137_{10} = 10001001_2 \quad \text{(binary)}$$
$$137_{10} = 0001\ 0011\ 0111 \quad \text{(BCD)}$$

The BCD code requires 12 bits, while the straight binary code requires only eight bits to represent 137. BCD requires more bits than straight binary to represent decimal numbers of more than one digit because BCD does not use all possible four-bit groups, as pointed out earlier, and is therefore somewhat inefficient.

The main advantage of the BCD code is the relative ease of converting to and from decimal. Only the four-bit code groups for the decimal digits 0 through 9 need to be remembered. This ease of conversion is especially important from a hardware standpoint because in a digital system, it is the logic circuits that perform the conversions to and from decimal.

REVIEW QUESTIONS

1. Represent the decimal value 178 by its straight binary equivalent. Then encode the same decimal number using BCD.
2. How many bits are required to represent an eight-digit decimal number in BCD?
3. What is an advantage of encoding a decimal number in BCD rather than in straight binary? What is a disadvantage?

2-5 THE GRAY CODE

Digital systems operate at very fast speeds and respond to changes that occur in the digital inputs. Just as in life, when multiple input conditions are changing at the same time, the situation can be misinterpreted and cause an erroneous reaction. When you look at the bits in a binary count sequence, it is clear that there are often several bits that must change states at the same time. For example, consider when the three-bit binary number for 3 changes to 4: all three bits must change state.

In order to reduce the likelihood of a digital circuit misinterpreting a changing input, the **Gray code** has been developed as a way to represent a sequence of numbers. The unique aspect of the Gray code is that only one bit ever changes between two successive numbers in the sequence. Table 2-2 shows the translation between three-bit binary and Gray code values. To convert binary to Gray, simply start on the most significant bit and use it as the Gray MSB as shown in Figure 2-2(a). Now compare the MSB binary with the next binary bit (B1). If they are the same, then $G1 = 0$. If they are different, then $G1 = 1$. G0 can be found by comparing B1 with B0.

TABLE 2-2 Three-bit binary and Gray code equivalents.

B_2	B_1	B_0	G_2	G_1	G_0
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	1
0	1	1	0	1	0
1	0	0	1	1	0
1	0	1	1	1	1
1	1	0	1	0	1
1	1	1	1	0	0

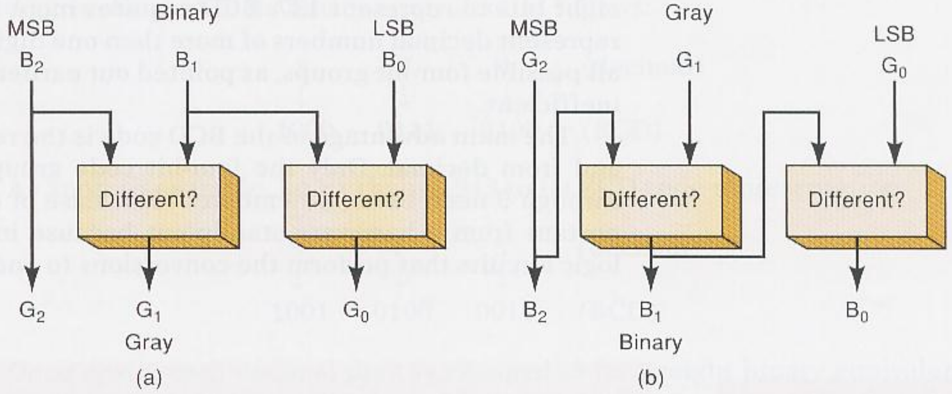
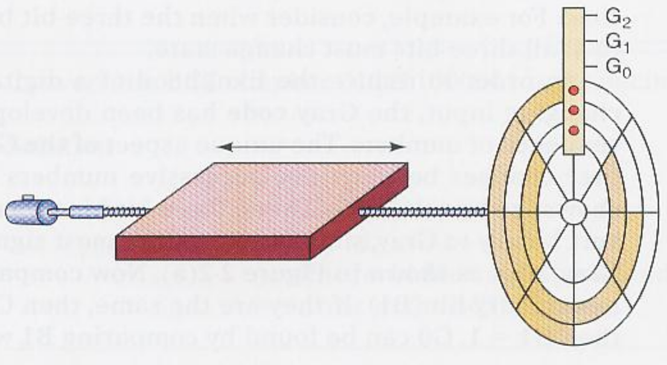

Binary
MSB
B_2
B_1
LSB
B_0
Different?
Different?
G_2
G_1
G_0
Gray
(a)
Gray
MSB
G_2
G_1
LSB
G_0
Different?
Different?
B_2
B_1
B_0
Binary
(b)

FIGURE 2-2 Converting (a) binary to Gray and (b) Gray to binary.

Conversion from Gray code back into binary is shown in Figure 2-2(b). Note that the MSB in Gray is always the same as the MSB in binary. The next binary bit is found by comparing the *binary* bit to the left with the *corresponding Gray code bit*. Similar bits produce a 0 and differing bits produce a 1. The most common application of the Gray code is in shaft position encoders as shown in Figure 2-3. These devices produce a binary value that represents the position of a rotating mechanical shaft. A practical shaft encoder would use many more bits than just three and divide the rotation into many more segments than eight, so that it could detect much smaller increments of rotation.

FIGURE 2-3 An eight-position, three-bit shaft encoder.



REVIEW QUESTIONS

1. Convert the number 0101 (binary) to its Gray code equivalent.
2. Convert 0101 (Gray code) to its binary number equivalent.

2-6 PUTTING IT ALL TOGETHER

Table 2-3 gives the representation of the decimal numbers 1 through 15 in the binary and hex number systems and also in the BCD and Gray codes. Examine it carefully and make sure you understand how it was obtained. Especially note how the BCD representation always uses four bits for each decimal digit.

TABLE 2-3

Decimal	Binary	Hexadecimal	BCD	GRAY
0	0	0	0000	0000
1	1	1	0001	0001
2	10	2	0010	0011
3	11	3	0011	0010
4	100	4	0100	0110
5	101	5	0101	0111
6	110	6	0110	0101
7	111	7	0111	0100
8	1000	8	1000	1100
9	1001	9	1001	1101
10	1010	A	0001 0000	1111
11	1011	B	0001 0001	1110
12	1100	C	0001 0010	1010
13	1101	D	0001 0011	1011
14	1110	E	0001 0100	1001
15	1111	F	0001 0101	1000

2-7 THE BYTE, NIBBLE, AND WORD

Bytes

Most microcomputers handle and store binary data and information in groups of eight bits, so a special name is given to a string of eight bits: it is called a **byte.** A byte always consists of eight bits, and it can represent any of numerous types of data or information. The following examples will illustrate.

EXAMPLE 2-8

How many bytes are in a 32-bit string (a string of 32 bits)?

Solution

32/8 = 4, so there are **four** bytes in a 32-bit string.

EXAMPLE 2-9

What is the largest decimal value that can be represented in binary using two bytes?

Solution

Two bytes is 16 bits, so the largest binary value will be equivalent to decimal $2^{16} - 1 = 65,535$.

EXAMPLE 2-10

How many bytes are needed to represent the decimal value 846,569 in BCD?

Solution

Each decimal digit converts to a four-bit BCD code. Thus, a six-digit decimal number requires 24 bits. These 24 bits are equal to **three** bytes. This is diagrammed below.

$$\begin{array}{cl} 8\ 4\ 6\ 5\ 6\ 9 & \text{(decimal)} \\ \underbrace{1000\ 0100}_{\text{byte1}}\ \underbrace{0110\ 0101}_{\text{byte2}}\ \underbrace{0110\ 1001}_{\text{byte3}} & \text{(BCD)} \end{array}$$

Nibbles

Binary numbers are often broken down into groups of four bits, as we have seen with BCD codes and hexadecimal number conversions. In the early days of digital systems, a term caught on to describe a group of four bits. Because it is half as big as a byte, it was named a **nibble**. The following examples illustrate the use of this term.

EXAMPLE 2-11

How many nibbles are in a byte?

Solution

2

EXAMPLE 2-12

What is the hex value of the least significant nibble of the binary number 1001 0101?

Solution

1001 0101

The least significant nibble is 0101 = 5.

Words

Bits, nibbles, and bytes are terms that represent a fixed number of binary digits. As systems have grown over the years, their capacity (appetite?) for handling binary data has also grown. A **word** is a group of bits that represents a certain unit of information. The size of the word depends on the size of the data pathway in the system that uses the information. The **word size** can be defined as the number of bits in the binary word that a digital system operates on. For example, the computer in your microwave oven can probably handle only one byte at a time. It has a word size of eight bits. On the other hand, the personal computer on your desk can handle eight bytes at a time, so it has a word size of 64 bits.

REVIEW QUESTIONS

1. How many bytes are needed to represent 235_{10} in binary?
2. What is the largest decimal value that can be represented in BCD using two bytes?
3. How many hex digits can a nibble represent?
4. How many nibbles are in one BCD digit?

2-8 ALPHANUMERIC CODES

In addition to numerical data, a computer must be able to handle nonnumerical information. In other words, a computer should recognize codes that represent letters of the alphabet, punctuation marks, and other special characters as well as numbers. These codes are called **alphanumeric codes**. A complete alphanumeric code would include the 26 lowercase letters, 26 uppercase letters, 10 numeric digits, 7 punctuation marks, and anywhere from 20 to 40 other characters, such as +, /, #, %, *, and so on. We can say that an alphanumeric code represents all of the various characters and functions that are found on a computer keyboard.

ASCII Code

The most widely used alphanumeric code is the **American Standard Code for Information Interchange (ASCII)**. The ASCII (pronounced "askee") code is a seven-bit code, and so it has $2^7 = 128$ possible code groups. This is more than enough to represent all of the standard keyboard characters as well as the control functions such as the (RETURN) and (LINEFEED) functions. Table 2-4 shows a listing of the standard seven-bit ASCII code. The table gives the hexadecimal and decimal equivalents. The seven-bit binary code for each character can be obtained by converting the hex value to binary.

EXAMPLE 2-13

Use Table 2-4 to find the seven-bit ASCII code for the backslash character (\).

Solution

The hex value given in Table 2-4 is 5C. Translating each hex digit into four-bit binary produces 0101 1100. The lower seven bits represent the ASCII code for \, or 1011100.

TABLE 2-4 Standard ASCII codes.

Character	HEX	Decimal	Character	HEX	Decimal	Character	HEX	Decimal	Character	HEX	Decimal
NUL (null)	0	0	Space	20	32	@	40	64	`	60	96
Start Heading	1	1	!	21	33	A	41	65	a	61	97
Start Text	2	2	“	22	34	B	42	66	b	62	98
End Text	3	3	#	23	35	C	43	67	c	63	99
End Transmit.	4	4	$	24	36	D	44	68	d	64	100
Enquiry	5	5	%	25	37	E	45	69	e	65	101
Acknowlege	6	6	&	26	38	F	46	70	f	66	102
Bell	7	7	`	27	39	G	47	71	g	67	103
Backspace	8	8	(	28	40	H	48	72	h	68	104
Horiz. Tab	9	9	)	29	41	I	49	73	i	69	105
Line Feed	A	10	*	2A	42	J	4A	74	j	6A	106
Vert. Tab	B	11	+	2B	43	K	4B	75	k	6B	107
Form Feed	C	12	,	2C	44	L	4C	76	l	6C	108
Carriage Return	D	13	-	2D	45	M	4D	77	m	6D	109
Shift Out	E	14	.	2E	46	N	4E	78	n	6E	110
Shift In	F	15	/	2F	47	O	4F	79	o	6F	111
Data Link Esc	10	16	0	30	48	P	50	80	p	70	112
Direct Control 1	11	17	1	31	49	Q	51	81	q	71	113
Direct Control 2	12	18	2	32	50	R	52	82	r	72	114
Direct Control 3	13	19	3	33	51	S	53	83	s	73	115
Direct Control 4	14	20	4	34	52	T	54	84	t	74	116
Negative ACK	15	21	5	35	53	U	55	85	u	75	117
Synch Idle	16	22	6	36	54	V	56	86	v	76	118
End Trans Block	17	23	7	37	55	W	57	87	w	77	119
Cancel	18	24	8	38	56	X	58	88	x	78	120
End of Medium	19	25	9	39	57	Y	59	89	y	79	121
Substitue	1A	26	:	3A	58	Z	5A	90	z	7A	122
Escape	1B	27	;	3B	59	[	5B	91	{	7B	123
Form separator	1C	28	<	3C	60	\	5C	92	|	7C	124
Group separator	1D	29	=	3D	61	]	5D	93	}	7D	125
Record Separator	1E	30	>	3E	62	^	5E	94	~	7E	126
Unit Separator	1F	31	?	3F	63	_	5F	95	Delete	7F	127

The ASCII code is used for the transfer of alphanumeric information between a computer and the external devices such as a printer or another computer. A computer also uses ASCII internally to store the information that an operator types in at the computer's keyboard. The following example illustrates this.

EXAMPLE 2-14

An operator is typing in a C language program at the keyboard of a certain microcomputer. The computer converts each keystroke into its ASCII code and stores the code as a byte in memory. Determine the binary strings that will be entered into memory when the operator types in the following C statement:

```
if (x>3)
```

Solution

Locate each character (including the space) in Table 2-4 and record its ASCII code.

i	69	0110	1001
f	66	0110	0110
space	20	0010	0000
(	28	0010	1000
x	78	0111	1000
>	3E	0011	1110
3	33	0011	0011
)	29	0010	1001

Note that a 0 was added to the leftmost bit of each ASCII code because the codes must be stored as bytes (eight bits). This adding of an extra bit is called *padding with 0s.*

REVIEW QUESTIONS

1. Encode the following message in ASCII code using the hex representation: "COST = $72."
2. The following padded ASCII-coded message is stored in successive memory locations in a computer:

   01010011 01010100 01001111 01010000

   What is the message?

2-9 PARITY METHOD FOR ERROR DETECTION

The movement of binary data and codes from one location to another is the most frequent operation performed in digital systems. Here are just a few examples:

- The transmission of digitized voice over a microwave link
- The storage of data in and retrieval of data from external memory devices such as magnetic and optical disk
- The transmission of digital data from a computer to a remote computer over telephone lines (i.e., using a modem). This is one of the major ways of sending and receiving information on the Internet.

Whenever information is transmitted from one device (the transmitter) to another device (the receiver), there is a possibility that errors can occur such that the receiver does not receive the identical information that was sent by the transmitter. The major cause of any transmission errors is *electrical noise,* which consists of spurious fluctuations in voltage or current that are present in all electronic systems to varying degrees. Figure 2-4 is a simple illustration of a type of transmission error.

The transmitter sends a relatively noise-free serial digital signal over a signal line to a receiver. However, by the time the signal reaches the receiver,

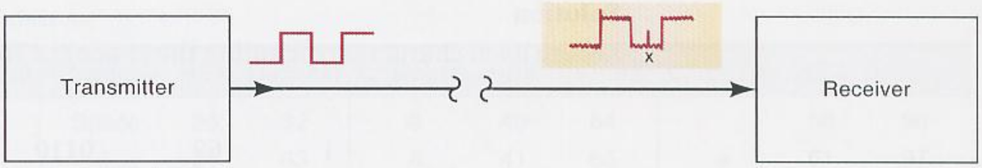


FIGURE 2-4 Example of noise causing an error in the transmission of digital data.

it contains a certain degree of noise superimposed on the original signal. Occasionally, the noise is large enough in amplitude that it will alter the logic level of the signal, as it does at point *x*. When this occurs, the receiver may incorrectly interpret that bit as a logic 1, which is not what the transmitter has sent.

Most modern digital equipment is designed to be relatively error-free, and the probability of errors such as the one shown in Figure 2-4 is very low. However, we must realize that digital systems often transmit thousands, even millions, of bits per second, so that even a very low rate of occurrence of errors can produce an occasional error that might prove to be bothersome, if not disastrous. For this reason, many digital systems employ some method for detection (and sometimes correction) of errors. One of the simplest and most widely used schemes for error detection is the **parity method**.

Parity Bit

A **parity bit** is an extra bit that is attached to a code group that is being transferred from one location to another. The parity bit is made either 0 or 1, depending on the number of 1s that are contained in the code group. Two different methods are used.

In the *even-parity* method, the value of the parity bit is chosen so that the total number of 1s in the code group (including the parity bit) is an *even* number. For example, suppose that the group is 1000011. This is the ASCII character "C." The code group has *three* 1s. Therefore, we will add a parity bit of 1 to make the total number of 1s an even number. The *new* code group, *including the parity bit,* thus becomes

$$\underset{\uparrow\text{—— added parity bit*}}{1}\ 1\,0\,0\,0\,0\,1\,1$$

If the code group contains an even number of 1s to begin with, the parity bit is given a value of 0. For example, if the code group were 1000001 (the ASCII code for "A"), the assigned parity bit would be 0, so that the new code, *including the parity bit,* would be 01000001.

The *odd-parity* method is used in exactly the same way except that the parity bit is chosen so the total number of 1s (including the parity bit) is an *odd* number. For example, for the code group 1000001, the assigned parity bit would be a 1. For the code group 1000011, the parity bit would be a 0.

Regardless of whether even parity or odd parity is used, the parity bit becomes an actual part of the code word. For example, adding a parity bit to the seven-bit ASCII code produces an eight-bit code. Thus, the parity bit is treated just like any other bit in the code.

---

*The parity bit can be placed at either end of the code group, but it is usually placed to the left of the MSB.

The parity bit is issued to detect any *single-bit* errors that occur during the transmission of a code from one location to another. For example, suppose that the character "A" is being transmitted and *odd* parity is being used. The transmitted code would be

1 1 0 0 0 0 0 1

When the receiver circuit receives this code, it will check that the code contains an odd number of 1s (including the parity bit). If so, the receiver will assume that the code has been correctly received. Now, suppose that because of some noise or malfunction the receiver actually receives the following code:

1 1 0 0 0 0 0 0

The receiver will find that this code has an *even* number of 1s. This tells the receiver that there must be an error in the code because presumably the transmitter and receiver have agreed to use odd parity. There is no way, however, that the receiver can tell which bit is in error because it does not know what the code is supposed to be.

It should be apparent that this parity method would not work if *two* bits were in error, because two errors would not change the "oddness" or "evenness" of the number of 1s in the code. In practice, the parity method is used only in situations where the probability of a single error is very low and the probability of double errors is essentially zero.

When the parity method is being used, the transmitter and the receiver must have agreement, in advance, as to whether odd or even parity is being used. There is no advantage of one over the other, although even parity seems to be used more often. The transmitter must attach an appropriate parity bit to each unit of information that it transmits. For example, if the transmitter is sending ASCII-coded data, it will attach a parity bit to each seven-bit ASCII code group. When the receiver examines the data that it has received from the transmitter, it checks each code group to see that the total number of 1s (including the parity bit) is consistent with the agreed-upon type of parity. This is often called *checking the parity* of the data. In the event that it detects an error, the receiver may send a message back to the transmitter asking it to retransmit the last set of data. The exact procedure that is followed when an error is detected depends on the particular system.

EXAMPLE 2-15

Computers often communicate with other remote computers over telephone lines. For example, this is how dial-up communication over the internet takes place. When one computer is transmitting a message to another, the information is usually encoded in ASCII. What actual bit strings would a computer transmit to send the message HELLO, using ASCII with even parity?

Solution

First, look up the ASCII codes for each character in the message. Then for each code, count the number of 1s. If it is an even number, attach a 0 as the

MSB. If it is an odd number, attach a 1. Thus, the resulting eight-bit codes (bytes) will all have an even number of 1s (including parity).

```
        attached even-parity bits
        ↓
H  =    0  1 0 0 1 0 0 0
E  =    1  1 0 0 0 1 0 1
L  =    1  1 0 0 1 1 0 0
L  =    1  1 0 0 1 1 0 0
O  =    1  1 0 0 1 1 1 1
```

REVIEW QUESTIONS

1. Attach an odd-parity bit to the ASCII code for the $ symbol, and express the result in hexadecimal.
2. Attach an even-parity bit to the BCD code for decimal 69.
3. Why can't the parity method detect a double error in transmitted data?

2-10 APPLICATIONS

Here are several applications that will serve as a review of some of the concepts covered in this chapter. These applications should give a sense of how the various number systems and codes are used in the digital world. More applications are presented in the end-of-chapter problems.

APPLICATION 2-1

A typical CD-ROM can store 650 megabytes of digital data. Since mega = 2^{20}, how many bits of data can a CD-ROM hold?

Solution

Remember that a byte is eight bits. Therefore, 650 megabytes is $650 \times 2^{20} \times 8 = 5{,}452{,}595{,}200$ bits.

APPLICATION 2-2

In order to program many microcontrollers, the binary instructions are stored in a file on a personal computer in a special way known as Intel Hex Format. The hexadecimal information is encoded into ASCII characters so it can be displayed easily on the PC screen, printed, and easily transmitted one character at a time over a standard PC's serial COM port. One line of an Intel Hex Format file is shown below:

```
:10002000F7CFFFCF1FEF2FEF2A95F1F71A95D9F7EA
```

The first character sent is the ASCII code for a colon, followed by a 1. Each has an even parity bit appended as the most significant bit. A test instrument captures the binary bit pattern as it goes across the cable to the microcontroller.

(a) What should the binary bit pattern (including parity) look like? (MSB – LSB)

(b) The value 10, following the colon, represents the total hexadecimal number of bytes that are to be loaded into the micro's memory. What is the decimal number of bytes being loaded?

(c) The number 0020 is a four-digit hex value representing the address where the first byte is to be stored. What is the biggest address possible? How many bits would it take to represent this address?

(d) The value of the first data byte is F7. What is the value (in binary) of the least significant nibble of this byte?

FFFF 1111 1111 1111 1111 16 bits

Solution

(a) ASCII codes are 3A (for :) and 31 (for 1) 00111010 10110001

even parity bit $\longrightarrow$ ↑ ↑

(b) 10 hex $= 1 \times 16 + 0 \times 1 = 16$ decimal bytes

(c) FFFF is the biggest possible value. Each hex digit is 4 bits, so we need 16 bits.

(d) The least significant nibble (4 bits) is represented by hex 7. In binary this would be 0111.

APPLICATION 2-3

A small process-control computer uses hexadecimal codes to represent its 16-bit memory addresses.

(a) How many hex digits are required?

(b) What is the range of addresses in hex?

(c) How many memory locations are there?

Solution

(a) Since 4 bits convert to a single hex digit, $16/4 = 4$ hex digits are needed.

(b) The binary range is 0000000000000000_2 to 1111111111111111_2. In hex, this becomes 0000_{16} to $FFFF_{16}$.

(c) With 4 hex digits, the total number of addresses is $16^4 = 65{,}536$.

APPLICATION 2-4

Numbers are entered into a microcontroller-based system in BCD, but stored in straight binary. As a programmer, you must decide whether you need a one-byte or two-byte storage location.

(a) How many bytes do you need if the system takes a two-digit decimal entry?

(b) What if you needed to be able to enter three digits?

Solution

(a) With two digits you can enter values up to 99 ($1001\ 1001_{BCD}$). In binary this value is 01100011, which will fit into an eight-bit memory location. Thus you can use a single byte.

(b) Three digits can represent up to 999 (1001 1001 1001). In binary this value is 1111100111 (10 bits). Thus you cannot use a single byte; you need two bytes.

APPLICATION 2-5

When ASCII characters must be transmitted between two independent systems (such as between a computer and a modem), there must be a way of telling the receiver when a new character is coming in. There is often a need to detect errors in the transmission as well. The method of transfer is called asynchronous data communication. The normal resting state of the transmission line is logic 1. When the transmitter sends an ASCII character, it must be "framed" so the receiver knows where the data begins and ends. The first bit must always be a start bit (logic 0). Next the ASCII code is sent LSB first and MSB last. After the MSB, a parity bit is appended to check for transmission errors. Finally, the transmission is ended by sending a stop bit (logic 1). A typical asynchronous transmission of a seven-bit ASCII code for the pound sign # (23 Hex) with even parity is shown in Figure 2-5.

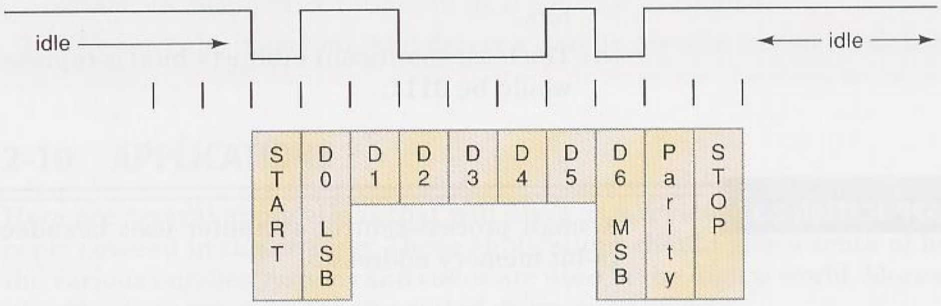


FIGURE 2-5 Asynchronous serial data with even parity.

SUMMARY

1. The hexadecimal number system is used in digital systems and computers as an efficient way of representing binary quantities.
2. In conversions between hex and binary, each hex digit corresponds to four bits.
3. The repeated-division method is used to convert decimal numbers to binary or hexadecimal.
4. Using an N-bit binary number, we can represent decimal values from 0 to $2^N - 1$.
5. The BCD code for a decimal number is formed by converting each digit of the decimal number to its four-bit binary equivalent.
6. The Gray code defines a sequence of bit patterns in which only one bit changes between successive patterns in the sequence.
7. A byte is a string of eight bits. A nibble is four bits. The word size depends on the system.
8. An alphanumeric code is one that uses groups of bits to represent all of the various characters and functions that are part of a typical computer's keyboard. The ASCII code is the most widely used alphanumeric code.
9. The parity method for error detection attaches a special parity bit to each transmitted group of bits.

IMPORTANT TERMS

- hexadecimal number system
- straight binary coding
- binary-coded-decimal (BCD) code
- Gray code
- byte
- nibble
- word
- word size
- alphanumeric code
- American Standard Code for Information Interchange (ASCII)
- parity method
- parity bit

PROBLEMS

SECTIONS 2-1 AND 2-2

2-1. Convert these binary numbers to decimal.

(a)*10110
(b) 10010101
(c)*100100001001
(d) 01101011
(e)*11111111
(f) 01101111
(g)*1111010111
(h) 11011111

2-2. Convert the following decimal values to binary.

(a)*37
(b) 13
(c)*189
(d) 1000
(e)*77
(f) 390
(g)*205
(h) 2133
(i)* 511

2-3. What is the largest decimal value that can be represented by (a)* an eight-bit binary number? (b) A 16-bit number?

SECTION 2-4

2-4. Convert each hex number to its decimal equivalent.

(a)*743
(b) 36
(c)*37FD
(d) 2000
(e)* 165
(f) ABCD
(g)* 7FF
(h) 1204

2-5. Convert each of the following decimal numbers to hex.

(a)*59
(b) 372
(c)*919
(d) 1024
(e)* 771
(f) 2313
(g)* 65,536
(h) 255

2-6. Convert each of the hex values from Problem 2-4 to binary.

2-7. Convert the binary numbers in Problem 2-1 to hex.

2-8. List the hex numbers in sequence from 195_{16} to 280_{16}.

2-9. When a large decimal number is to be converted to binary, it is sometimes easier to convert it first to hex, and then from hex to binary. Try this procedure for 2133_{10} and compare it with the procedure used in Problem 2-2(h).

2-10. How many hex digits are required to represent decimal numbers up to 20,000?

2-11. Convert these hex values to decimal.

(a)*92
(b) 1A6
(c)*37FD
(d) ABCD
(e)* 000F
(f) 55
(g)* 2C0
(h) 7FF

---

*Answers to problems marked with an asterisk can be found in the back of the text.

2-12. Convert these decimal values to hex.
(a)*75
(b) 314
(c)*2048
(d) 24
(e)*7245
(f) 498
(g)*25,619
(h) 4095

2-13. Take each four-bit binary number in the order they are written and write the equivalent hex digit without performing a calculation by hand or by calculator.
(a) 1001
(b) 1101
(c) 1000
(d) 0000
(e) 1111
(f) 0010
(g) 1010
(h) 1001
(i) 1011
(j) 1100
(k) 0011
(l) 0100
(m) 0001
(n) 0101
(o) 0111
(p) 0110

2-14. Take each hex digit and write its four-bit binary value without performing any calculations by hand or by calculator.
(a) 6
(b) 7
(c) 5
(d) 1
(e) 4
(f) 3
(g) C
(h) B
(i) 9
(j) A
(k) 2
(l) F
(m) 0
(n) 8
(o) D
(p) 9

2-15.* Convert the binary numbers in Problem 2-1 to hexadecimal.

2-16.* Convert the hex values in Problem 2-11 to binary.

2-17.* List the hex numbers in sequence from 280 to 2A0.

2-18. How many hex digits are required to represent decimal numbers up to 1 million?

SECTION 2-5

2-19. Encode these decimal numbers in BCD.
(a)*47
(b) 962
(c)*187
(d) 6727
(e)*13
(f) 529
(g)*89,627
(h) 1024

2-20. How many bits are required to represent the decimal numbers in the range from 0 to 999 using (a) straight binary code? (b) Using BCD code?

2-21. The following numbers are in BCD. Convert them to decimal.
(a)*1001011101010010
(b) 000110000100
(c)*011010010101
(d) 0111011101110101
(e)*010010010010
(f) 010101010101

SECTION 2-7

2-22.*(a) How many bits are contained in eight bytes?
(b) What is the largest hex number that can be represented in four bytes?
(c) What is the largest BCD-encoded decimal value that can be represented in three bytes?

2-23. (a) Refer to Table 2-4. What is the most significant nibble of the ASCII code for the letter X?
(b) How many nibbles can be stored in a 16-bit word?
(c) How many bytes does it take to make up a 24-bit word?

SECTIONS 2-8 AND 2-9

2-24. Represent the statement "X = 3 × Y" in ASCII code. Attach an odd-parity bit.

2-25.* Attach an *even*-parity bit to each of the ASCII codes for Problem 2-24, and give the results in hex.

2-26. The following bytes (shown in hex) represent a person's name as it would be stored in a computer's memory. Each byte is a padded ASCII code. Determine the name of each person.

(a)* 42 45 4E 20 53 4D 49 54 48
(b) 4A 6F 65 20 47 72 65 65 6E

2-27. Convert the following decimal numbers to BCD code and then attach an *odd* parity bit.

(a)* 74	(c)* 8884	(e)* 165
(b) 38	(d) 275	(f) 9201

2-28.* In a certain digital system, the decimal numbers from 000 through 999 are represented in BCD code. An *odd*-parity bit is also included at the end of each code group. Examine each of the code groups below, and assume that each one has just been transferred from one location to another. Some of the groups contain errors. Assume that *no more than* two errors have occurred for each group. Determine which of the code groups have a single error and which of them *definitely* have a double error. (*Hint:* Remember that this is a BCD code.)

(a) $\underset{\uparrow}{1}001010110000$ parity bit
(b) 0100011101100
(c) 0111110000011
(d) 1000011000101

2-29. Suppose that the receiver received the following data from the transmitter of Example 2-16:

0 1 0 0 1 0 0 0
1 1 0 0 0 1 0 1
1 1 0 0 1 1 0 0
1 1 0 0 1 0 0 0
1 1 0 0 1 1 0 0

What errors can the receiver determine in these received data?

DRILL QUESTIONS

2-30.* Perform each of the following conversions. For some of them, you may want to try several methods to see which one works best for you. For example, a binary-to-decimal conversion may be done directly, or it may be done as a binary-to-hex conversion followed by a hex-to-decimal conversion.

(a) $1417_{10} = \underline{\qquad}_2$
(b) $255_{10} = \underline{\qquad}_2$
(c) $11010001_2 = \underline{\qquad}_{10}$
(d) $1110101000100111_2 = \underline{\qquad}_{10}$

(e) 2497_{10} = _____$_{16}$
(f) 511_{10} = _____ (BCD)
(g) 235_{16} = _____$_{10}$
(h) 4316_{10} = _____$_{16}$
(i) $7A9_{16}$ = _____$_{10}$
(j) $3E1C_{16}$ = _____$_{10}$
(k) 1600_{10} = _____$_{16}$
(l) $38{,}187_{10}$ = _____$_{16}$
(m) 865_{10} = _____ (BCD)
(n) 100101000111 (BCD) = _____$_{10}$
(o) 465_{16} = _____$_{2}$
(p) $B34_{16}$ = _____$_{2}$
(q) 01110100 (BCD) = _____$_{2}$
(r) 111010_2 = _____ (BCD)

2-31.*Represent the decimal value 37 in each of the following ways.

(a) straight binary
(b) BCD
(c) hex
(d) ASCII (i.e., treat each digit as a character)

2-32.*Fill in the blanks with the correct word or words.

(a) Conversion from decimal to _____ requires repeated division by 16.

(b) Conversion from decimal to binary requires repeated division by _____.

(c) In the BCD code, each _____ is converted to its four-bit binary equivalent.

(d) The _____ code has the characteristic that only one bit changes in going from one step to the next.

(e) A transmitter attaches a _____ to a code group to allow the receiver to detect _____.

(f) The _____ code is the most common alphanumeric code used in computer systems.

(g) _____ is often used as a convenient way to represent large binary numbers.

(h) A string of eight bits is called a _____.

2-33. Write the binary number that results when each of the following numbers is incremented by one.

(a)*0111 (b) 010011 (c) 1011

2-34. Decrement each binary number.

(a)*1110 (b) 101000 (c) 1110

2-35. Write the number that results when each of the following is incremented.

(a)*7779_{16} (c)*$0FFF_{16}$ (e)*$9FF_{16}$
(b) 9999_{16} (d) 2000_{16} (f) $100A_{16}$

2-36.*Repeat Problem 2-35 for the decrement operation.

CHALLENGING EXERCISES

2-37.* In a microcomputer, the *addresses* of memory locations are binary numbers that identify each memory circuit where a byte is stored. The number of bits that make up an address depends on how many memory locations there are. Since the number of bits can be very large, the addresses are often specified in hex instead of binary.

(a) If a microcomputer uses a 20-bit address, how many different memory locations are there?

(b) How many hex digits are needed to represent the address of a memory location?

(c) What is the hex address of the 256th memory location? (*Note:* The first address is always 0.)

2-38. In an audio CD, the audio voltage signal is typically sampled about 44,000 times per second, and the value of each sample is recorded on the CD surface as a binary number. In other words, each recorded binary number represents a single voltage point on the audio signal waveform.

(a) If the binary numbers are six bits in length, how many different voltage values can be represented by a single binary number? Repeat for eight bits and ten bits.

(b) If ten-bit numbers are used, how many bits will be recorded on the CD in 1 second?

(c) If a CD can typically store 5 billion bits, how many seconds of audio can be recorded when ten-bit numbers are used?

2-39.* A black-and-white digital camera lays a fine grid over an image and then measures and records a binary number representing the level of gray it sees in each cell of the grid. For example, if four-bit numbers are used, the value of black is set to 0000 and the value of white to 1111, and any level of gray is somewhere between 0000 and 1111. If six-bit numbers are used, black is 000000, white is 111111, and all grays are between the two.

Suppose we wanted to distinguish among 254 different levels of gray within each cell of the grid. How many bits would we need to use to represent these levels?

2-40. A 3-Megapixel digital camera stores an eight-bit number for the brightness of each of the primary colors (red, green, blue) found in each picture element (pixel). If every bit is stored (no data compression), how many pictures can be stored on a 128-Megabyte memory card? (Note: In digital systems, Mega means 2^{20}.)

2-41. Construct a table showing the binary, hex, and BCD representations of all decimal numbers from 0 to 15. Compare your table with Table 2-3.

ANSWERS TO SECTION REVIEW QUESTIONS

SECTION 2-1

1. 2267 2. 32768

SECTION 2-2

1. 1010011 2. 1011011001 3. 20 bits

SECTION 2-3

1. 9422 2. C2D; 110000101101 3. 97B5 4. E9E, E9F, EA0, EA1
5. 11010100100111 6. 0 to 65,535

SECTION 2-4

1. 10110010_2; 000101111000 (BCD) 2. 32 3. Advantage: ease of conversion. Disadvantage: BCD requires more bits.

SECTION 2-5

1. 0111 2. 0110

SECTION 2-7

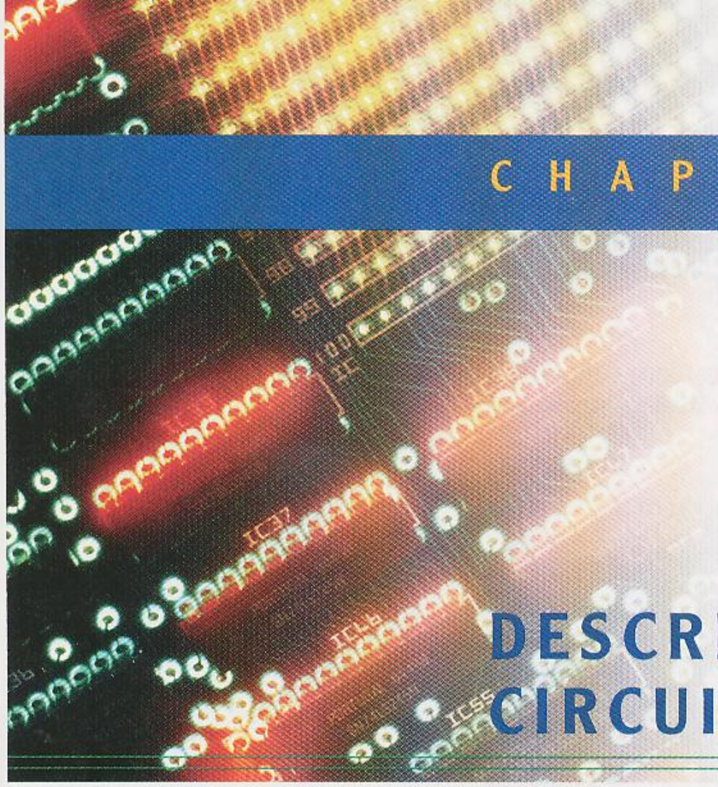
1. One 2. 9999 3. One 4. One

SECTION 2-8

1. 43, 4F, 53, 54, 20, 3D, 20, 24, 37, 32 2. STOP

SECTION 2-9

1. A4 2. 001101001 3. Two errors in the data would not change the oddness or evenness of the number of 1s in the data.



CHAPTER 3

DESCRIBING LOGIC CIRCUITS

OUTLINE

3-1 Boolean Constants and Variables
3-2 Truth Tables
3-3 OR Operation with OR Gates
3-4 AND Operation with AND Gates
3-5 NOT Operation
3-6 Describing Logic Circuits Algebraically
3-7 Evaluating Logic-Circuit Outputs
3-8 Implementing Circuits from Boolean Expressions
3-9 NOR Gates and NAND Gates
3-10 Boolean Theorems
3-11 DeMorgan's Theorems
3-12 Universality of NAND Gates and NOR Gates
3-13 Alternate Logic-Gate Representations
3-14 Which Gate Representation to Use
3-15 IEEE/ANSI Standard Logic Symbols
3-16 Summary of Methods to Describe Logic Circuits
3-17 Description Languages Versus Programming Languages
3-18 Implementing Logic Circuits with PLDs
3-19 HDL Format and Syntax
3-20 Intermediate Signals

OBJECTIVES

Upon completion of this chapter, you will be able to:

- Perform the three basic logic operations.
- Describe the operation of and construct the truth tables for the AND, NAND, OR, and NOR gates, and the NOT (INVERTER) circuit.
- Draw timing diagrams for the various logic-circuit gates.
- Write the Boolean expression for the logic gates and combinations of logic gates.
- Implement logic circuits using basic AND, OR, and NOT gates.
- Appreciate the potential of Boolean algebra to simplify complex logic circuits.
- Use DeMorgan's theorems to simplify logic expressions.
- Use either of the universal gates (NAND or NOR) to implement a circuit represented by a Boolean expression.
- Explain the advantages of constructing a logic-circuit diagram using the alternate gate symbols versus the standard logic-gate symbols.
- Describe the concept of active-LOW and active-HIGH logic signals.
- Draw and interpret the IEEE/ANSI standard logic-gate symbols.
- Use several methods to describe the operation of logic circuits.
- Interpret simple circuits defined by a hardware description language (HDL).
- Explain the difference between an HDL and a computer programming language.
- Create an HDL file for a simple logic gate.
- Create an HDL file for combinational circuits with intermediate variables.

INTRODUCTION

Chapters 1 and 2 introduced the concepts of logic levels and logic circuits. In logic, only two possible conditions exist for any input or output: true and false. The binary number system uses only two digits, 1 and 0, so it is perfect for representing logical relationships. Digital logic circuits use predefined voltage ranges to represent these binary states. Using these concepts, we can create circuits made of little more than processed beach sand and wire that make consistent, intelligent, logical decisions. It is vitally important that we have a method to describe the logical decisions made by these circuits. In other words, we must describe how they operate. In this chapter, we will discover many ways to describe their operation. Each description

method is important because all these methods commonly appear in technical literature and system documentation and are used in conjunction with modern design and development tools.

Life is full of examples of circumstances that are in one state or another. For example, a creature is either alive or dead, a light is either on or off, a door is locked or unlocked, and it is either raining or it is not. In 1854, a mathematician named George Boole wrote *An Investigation of the Laws of Thought*, in which he described the way we make logical decisions based on true or false circumstances. The methods he described are referred to today as Boolean logic, and the system of using symbols and operators to describe these decisions is called Boolean algebra. In the same way we use symbols such as x and y to represent unknown numerical values in regular algebra, Boolean algebra uses symbols to represent a logical expression that has one of two possible values: true or false. The logical expression might be *door is closed, button is pressed,* or *fuel is low*. Writing these expressions is very tedious, and so we tend to substitute symbols such as A, B, and C.

The main purpose of these logical expressions is to describe the relationship between a logic circuit's output (the decision) and its inputs (the circumstances). In this chapter, we will study the most basic logic circuits—*logic gates*—which are the fundamental building blocks from which all other logic circuits and digital systems are constructed. We will see how the operation of the different logic gates and the more complex circuits formed from combinations of logic gates can be described and analyzed using Boolean algebra. We will also get a glimpse of how Boolean algebra can be used to simplify a circuit's Boolean expression so that the circuit can be rebuilt using fewer logic gates and/or fewer connections. Much more will be done with circuit simplification in Chapter 4.

Boolean algebra is not only used as a tool for analysis and simplification of logic systems. It can also be used as a tool to create a logic circuit that will produce the desired input/output relationship. This process is often called synthesis of logic circuits as opposed to analysis. Other techniques have been used in the analysis, synthesis, and documentation of logic systems and circuits including truth tables, schematic symbols, timing diagrams, and—last but by no means least—language. To categorize these methods, we could say that Boolean algebra is a mathematic tool, truth tables are data organizational tools, schematic symbols are drawing tools, timing diagrams are graphing tools, and language is the universal description tool.

Today, any of these tools can be used to provide input to computers. The computers can be used to simplify and translate between these various forms of description and ultimately provide an output in the form necessary to implement a digital system. To take advantage of the powerful benefits of computer software, we must first fully understand the acceptable ways for describing these systems in terms the computer can understand. This chapter will lay the groundwork for further study of these vital tools for synthesis and analysis of digital systems.

Clearly the tools described here are invaluable tools in describing, analyzing, designing, and implementing digital circuits. The student who expects to work in the digital field must work hard at understanding and becoming comfortable with Boolean algebra (believe us, it's much, much easier than conventional algebra) and all the other tools. Do *all* of the examples, exercises, and problems, even the ones your instructor doesn't assign. When those run out, make up your own. The time you spend will be well worth it because you will see your skills improve and your confidence grow.

3-1 BOOLEAN CONSTANTS AND VARIABLES

Boolean algebra differs in a major way from ordinary algebra because Boolean constants and variables are allowed to have only two possible values, 0 or 1. A Boolean variable is a quantity that may, at different times, be equal to either 0 or 1. Boolean variables are often used to represent the voltage level present on a wire or at the input/output terminals of a circuit. For example, in a certain digital system, the Boolean value of 0 might be assigned to any voltage in the range from 0 to 0.8 V, while the Boolean value of 1 might be assigned to any voltage in the range 2 to 5 V.*

Thus, Boolean 0 and 1 do not represent actual numbers but instead represent the state of a voltage variable, or what is called its **logic level**. A voltage in a digital circuit is said to be at the logic 0 level or the logic 1 level, depending on its actual numerical value. In digital logic, several other terms are used synonymously with 0 and 1. Some of the more common ones are shown in Table 3-1. We will use the 0/1 and LOW/HIGH designations most of the time.

TABLE 3-1

Logic 0	Logic 1
False	True
Off	On
Low	High
No	Yes
Open switch	Closed switch

As we said in the introduction, **Boolean algebra** is a means for expressing the relationship between a logic circuit's inputs and outputs. The inputs are considered logic variables whose logic levels at any time determine the output levels. In all our work to follow, we shall use letter symbols to represent logic variables. For example, the letter A might represent a certain digital circuit input or output, and at any time we must have either $A = 0$ or $A = 1$: if not one, then the other.

Because only two values are possible, Boolean algebra is relatively easy to work with compared with ordinary algebra. In Boolean algebra, there are no fractions, decimals, negative numbers, square roots, cube roots, logarithms, imaginary numbers, and so on. In fact, in Boolean algebra there are only *three* basic operations: *OR, AND,* and *NOT.*

These basic operations are called *logic operations.* Digital circuits called *logic gates* can be constructed from diodes, transistors, and resistors connected so that the circuit output is the result of a basic logic operation *(OR, AND, NOT)* performed on the inputs. We will be using Boolean algebra first to describe and analyze these basic logic gates, then later to analyze and design combinations of logic gates connected as logic circuits.

3-2 TRUTH TABLES

A **truth table** is a means for describing how a logic circuit's output depends on the logic levels present at the circuit's inputs. Figure 3-1(a) illustrates a truth table for one type of two-input logic circuit. The table lists all possible

---

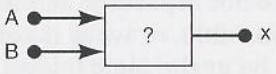
*Voltages between 0.8 and 2 V are undefined (neither 0 nor 1) and should not occur under normal circumstances.

FIGURE 3-1 Example truth tables for (a) two-input, (b) three-input, and (c) four-input circuits.

Inputs

Output

A	B	x
0	0	1
0	1	0
1	0	1
1	1	0



(a)

A	B	C	x
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

(b)

A	B	C	D	x
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

(c)

UTT - SAN FERNANDO CAMPUS LIBRARY

combinations of logic levels present at inputs A and B, along with the corresponding output level x. The first entry in the table shows that when A and B are both at the 0 level, the output x is at the 1 level or, equivalently, in the 1 state. The second entry shows that when input B is changed to the 1 state, so that $A = 0$ and $B = 1$, the output x becomes a 0. In a similar way, the table shows what happens to the output state for any set of input conditions.

Figures 3-1(b) and (c) show samples of truth tables for three- and four-input logic circuits. Again, each table lists all possible combinations of input logic levels on the left, with the resultant logic level for output x on the right. Of course, the actual values for x will depend on the type of logic circuit.

Note that there are 4 table entries for the two-input truth table, 8 entries for a three-input truth table, and 16 entries for the four-input truth table. The number of input combinations will equal 2^N for an N-input truth table. Also note that the list of all possible input combinations follows the binary counting sequence, and so it is an easy matter to write down all of the combinations without missing any.

REVIEW QUESTIONS

1. What is the output state of the four-input circuit represented in Figure 3-1(c) when all inputs except B are 1?
2. Repeat question 1 for the following input conditions: $A = 1, B = 0, C = 1, D = 0$.
3. How many table entries are needed for a five-input circuit?

3-3 OR OPERATION WITH OR GATES

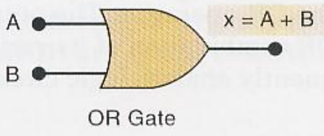
The **OR operation** is the first of the three basic Boolean operations to be learned. An example can be found in the kitchen oven. The light inside the oven should turn on if either the *oven light switch is on* OR if the *door is opened.* The letter A could be used to represent the *oven light switch is on* and B could represent *door is opened.* The letter x could represent the *light is on.* The truth table in Figure 3-2(a) shows what happens when two logic inputs, A and B, are combined using the OR operation to produce the output x. The table shows that x is a logic 1 for every combination of input levels where one *or* more inputs are 1. The only case where x is a 0 is when both inputs are 0.

FIGURE 3-2 (a) Truth table defining the OR operation; (b) circuit symbol for a two-input OR gate.

OR

A	B	x = A + B
0	0	0
0	1	1
1	0	1
1	1	1

(a)



(b)

The Boolean expression for the OR operation is

$$x = A + B$$

In this expression, the + sign does not stand for ordinary addition; it stands for the OR operation. The OR operation is similar to ordinary addition except for the case where A and B are both 1; the OR operation produces $1 + 1 = 1$, not $1 + 1 = 2$. In Boolean algebra, 1 is as high as we go, so we can never have a result greater than 1. The same holds true for combining three inputs using the OR operation. Here we have $x = A + B + C$. If we consider the case where all three inputs are 1, we have

$$x = 1 + 1 + 1 = 1$$

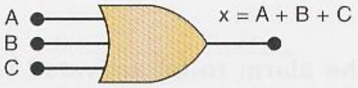
The expression $x = A + B$ is read as "x equals A OR B," which means that x will be 1 when A or B or both are 1. Likewise, the expression $x = A + B + C$ is read as "x equals A OR B OR C," which means that x will be 1 when A or B or C or any combination of them are 1. To describe this circuit in the English language we could say that *x is true (1)* ***WHEN*** *A is true (1)* **OR** *B is true (1)* **OR** *C is true (1).*

OR Gate

In digital circuitry, an **OR gate*** is a circuit that has two or more inputs and whose output is equal to the OR combination of the inputs. Figure 3-2(b) is the logic symbol for a two-input OR gate. The inputs A and B are logic voltage levels, and the output x is a logic voltage level whose value is the result of the OR operation on A and B; that is, $x = A + B$. In other words, the OR gate operates so that its output is HIGH (logic 1) if either input *A or B or both* are at a logic 1 level. The OR gate output will be LOW (logic 0) only if all its inputs are at logic 0.

This same idea can be extended to more than two inputs. Figure 3-3 shows a three-input OR gate and its truth table. Examination of this truth table shows again that the output will be 1 for every case where one or more inputs are 1. This general principle is the same for OR gates with any number of inputs.

FIGURE 3-3 Symbol and truth table for a three-input OR gate.



A	B	C	x = A + B + C
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

---

*The term *gate* comes from the inhibit/enable operation discussed in Chapter 4.

Using the language of Boolean algebra, the output x can be expressed as $x = A + B + C$, where again it must be emphasized that the $+$ represents the OR operation. The output of any OR gate, then, can be expressed as the OR combination of its various inputs. We will put this to use when we subsequently analyze logic circuits.

Summary of the OR Operation

The important points to remember concerning the OR operation and OR gates are:

1. The OR operation produces a result (output) of 1 whenever *any* input is a 1. Otherwise the output is 0.
2. An OR gate is a logic circuit that performs an OR operation on the circuit's inputs.
3. The expression $x = A + B$ is read as "x equals A OR B."

EXAMPLE 3-1

In many industrial control systems, it is required to activate an output function whenever any one of several inputs is activated. For example, in a chemical process it may be desired that an alarm be activated whenever the process temperature exceeds a maximum value *or* whenever the pressure goes above a certain limit. Figure 3-4 is a block diagram of this situation. The temperature transducer circuit produces an output voltage proportional to the process temperature. This voltage, V_T, is compared with a temperature reference voltage, V_{TR}, in a voltage comparator circuit. The comparator output, T_H, is normally a low voltage (logic 0), but it switches to a high voltage (logic 1) when V_T exceeds V_{TR}, indicating that the process temperature is too high. A similar arrangement is used for the pressure measurement, so that its associated comparator output, P_H, goes from LOW to HIGH when the pressure is too high.

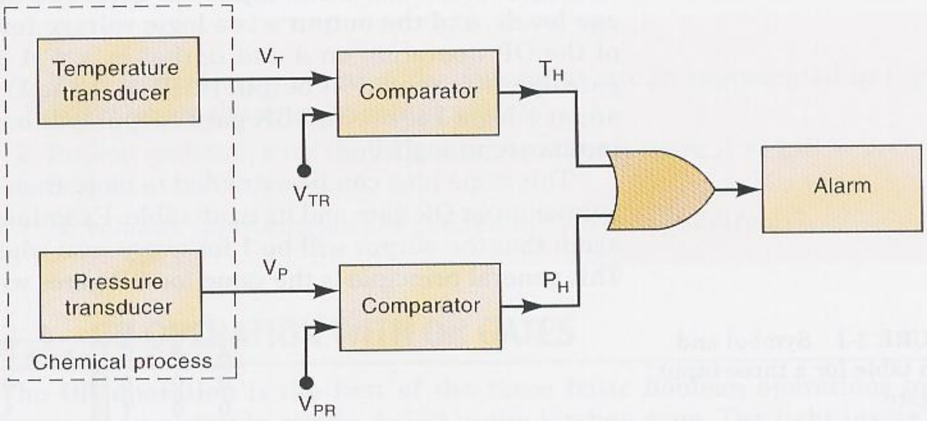


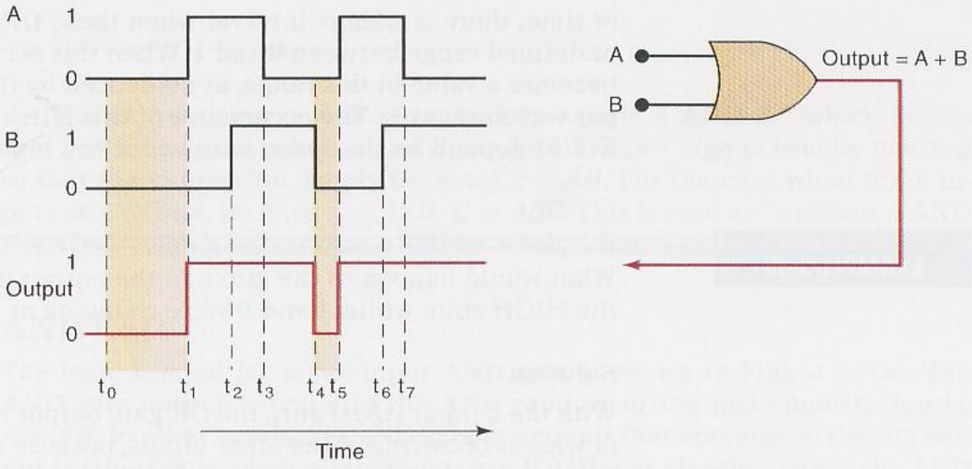
FIGURE 3-4 Example of the use of an OR gate in an alarm system.

Since we want the alarm to be activated when either temperature *or* pressure is too high, it should be apparent that the two comparator outputs can be fed to a two-input OR gate. The OR gate output thus goes HIGH (1) for either alarm condition and will activate the alarm. This same idea can obviously be extended to situations with more than two process variables.

EXAMPLE 3-2

Determine the OR gate output in Figure 3-5. The OR gate inputs A and B are varying according to the timing diagrams shown. For example, A starts out LOW at time t_0, goes HIGH at t_1, back to LOW at t_3, and so on.

FIGURE 3-5 Example 3-2.



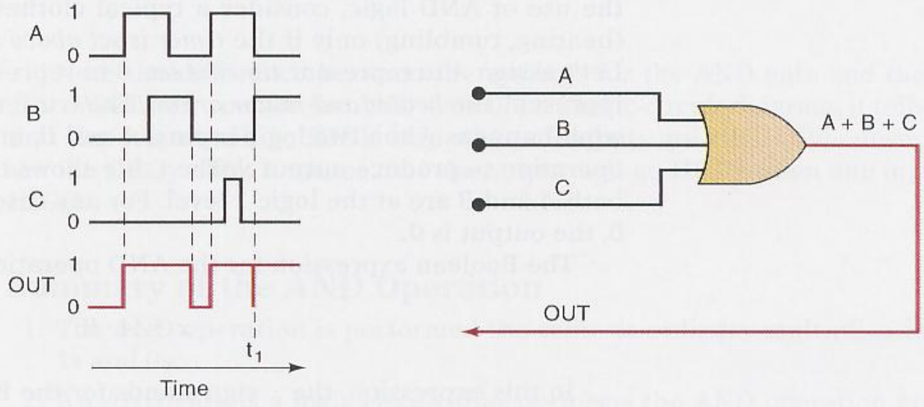
Solution

The OR gate output will be HIGH whenever *any* input is HIGH. Between time t_0 and t_1, both inputs are LOW, so OUTPUT = LOW. At t_1, input A goes HIGH while B remains LOW. This causes OUTPUT to go HIGH at t_1 and stay HIGH until t_4 because, during this interval, one or both inputs are HIGH. At t_4, input B goes from 1 to 0 so that now both inputs are LOW, and this drives OUTPUT back to LOW. At t_5, A goes HIGH, sending OUTPUT back HIGH, where it stays for the rest of the shown time span.

EXAMPLE 3-3A

For the situation depicted in Figure 3-6, determine the waveform at the OR gate output.

FIGURE 3-6 Examples 3-3A and B.



Solution

The three OR gate inputs A, B, and C are varying, as shown by their waveform diagrams. The OR gate output is determined by realizing that it will be

HIGH whenever *any* of the three inputs is at a HIGH level. Using this reasoning, the OR output waveform is as shown in the figure. Particular attention should be paid to what occurs at time t_1. The diagram shows that, at that instant of time, input A is going from HIGH to LOW while input B is going from LOW to HIGH. Since these inputs are making their transitions at approximately the same time, and since these transitions take a certain amount of time, there is a short interval when these OR gate inputs are both in the undefined range between 0 and 1. When this occurs, the OR gate output also becomes a value in this range, as evidenced by the glitch or spike on the output waveform at t_1. The occurrence of this glitch and its size (amplitude and width) depend on the speed with which the input transitions occur.

EXAMPLE 3-3B

What would happen to the glitch in the output in Figure 3-6 if input C sat in the HIGH state while A and B were changing at time t_1?

Solution

With the C input HIGH at t_1, the OR gate output will remain HIGH, regardless of what is occurring at the other inputs, because any HIGH input will keep an OR gate output HIGH. Therefore, the glitch will not appear in the output.

REVIEW QUESTIONS

1. What is the only set of input conditions that will produce a LOW output for any OR gate?
2. Write the Boolean expression for a six-input OR gate.
3. If the A input in Figure 3-6 is permanently kept at the 1 level, what will the resultant output waveform be?

3-4 AND OPERATION WITH AND GATES

The **AND operation** is the second basic Boolean operation. As an example of the use of AND logic, consider a typical clothes dryer. It is drying clothes (heating, tumbling) only if the *timer is set above zero* AND the *door is closed.* Let's assign A to represent *timer is set,* B to represent *door is closed,* and x can represent the *heater and motor are on.* The truth table in Figure 3-7(a) shows what happens when two logic inputs, A and B, are combined using the AND operation to produce output x. The table shows that x is a logic 1 only when both A and B are at the logic 1 level. For any case where one of the inputs is 0, the output is 0.

The Boolean expression for the AND operation is

$$x = A \cdot B$$

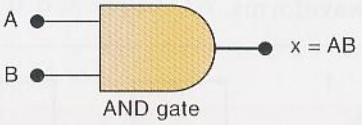
In this expression, the $\cdot$ sign stands for the Boolean AND operation and not the multiplication operation. However, the AND operation on Boolean variables operates the same as ordinary multiplication, as examination of the truth table shows, so we can think of them as being the same. This characteristic can be helpful when evaluating logic expressions that contain AND operations.

FIGURE 3-7 (a) Truth table for the AND operation; (b) AND gate symbol.

AND

A	B	x = A · B
0	0	0
0	1	0
1	0	0
1	1	1

(a)



(b)

The expression $x = A \cdot B$ is read as "x equals A AND B," which means that x will be 1 only when A and B are both 1. The $\cdot$ sign is usually omitted so that the expression simply becomes $x = AB$. For the case when three inputs are ANDed, we have $x = A \cdot B \cdot C = ABC$. This is read as "$x$ equals A AND B AND C," which means that x will be 1 only when A and B and C are all 1.

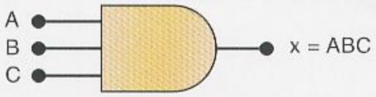
AND Gate

The logic symbol for a two-input **AND** gate is shown in Figure 3-7(b). The AND gate output is equal to the AND product of the logic inputs; that is, $x = AB$. In other words, the AND gate is a circuit that operates so that its output is HIGH only when all its inputs are HIGH. For all other cases, the AND gate output is LOW.

This same operation is characteristic of AND gates with more than two inputs. For example, a three-input AND gate and its accompanying truth table are shown in Figure 3-8. Once again, note that the gate output is 1 only for the case where $A = B = C = 1$. The expression for the output is $x = ABC$. For a four-input AND gate, the output is $x = ABCD$, and so on.

FIGURE 3-8 Truth table and symbol for a three-input AND gate.

A	B	C	x = ABC
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1



Note the difference between the symbols for the AND gate and the OR gate. Whenever you see the AND symbol on a logic-circuit diagram, it tells you that the output will go HIGH *only* when *all* inputs are HIGH. Whenever you see the OR symbol, it means that the output will go HIGH when *any* input is HIGH.

Summary of the AND Operation

1. The AND operation is performed the same as ordinary multiplication of 1s and 0s.
2. An AND gate is a logic circuit that performs the AND operation on the circuit's inputs.
3. An AND gate output will be 1 *only* for the case when *all* inputs are 1; for all other cases, the output will be 0.
4. The expression $x = AB$ is read as "x equals A AND B."

EXAMPLE 3-4

Determine the output x from the AND gate in Figure 3-9 for the given input waveforms.

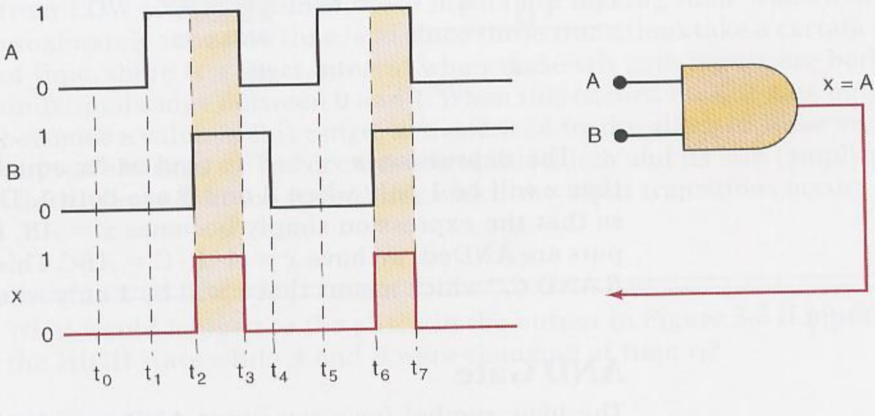


FIGURE 3-9 Example 3-4.

Solution

The output of an AND gate is determined by realizing that it will be HIGH only when all inputs are HIGH at the same time. For the input waveforms given, this condition is met only during intervals $t_2 - t_3$ and $t_6 - t_7$. At all other times, one or more of the inputs are 0, thereby producing a LOW output. Note that input level changes that occur while the other input is LOW have no effect on the output.

EXAMPLE 3-5A

Determine the output waveform for the AND gate shown in Figure 3-10.

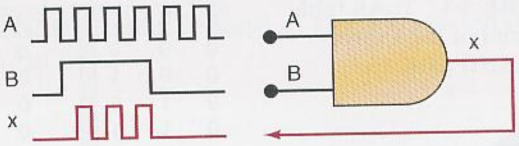


FIGURE 3-10 Examples 3-5A and B.

Solution

The output x will be at 1 only when A and B are both HIGH at the same time. Using this fact, we can determine the x waveform as shown in the figure.

Notice that the x waveform is 0 whenever B is 0, regardless of the signal at A. Also notice that whenever B is 1, the x waveform is the same as A. Thus, we can think of the B input as a *control* input whose logic level determines whether or not the A waveform gets through to the x output. In this situation, the AND gate is used as an *inhibit circuit*. We can say that $B = 0$ is the inhibit condition producing a 0 output. Conversely, $B = 1$ is the *enable* condition, which enables A to reach the output. This inhibit operation is an important application of AND gates, which will be encountered later.

EXAMPLE 3-5B

What will happen to the x output waveform in Figure 3-10 if the B input is kept at the 0 level?

Solution

With B kept LOW, the x output will also stay LOW. This can be reasoned in two different ways. First, with $B = 0$ we have $x = A \cdot B = A \cdot 0 = 0$ because anything multiplied (ANDed) by 0 will be 0. Another way to look at it is that an AND gate requires that all inputs be HIGH for the output to be HIGH, and this cannot happen if B is kept LOW.

REVIEW QUESTIONS

1. What is the only input combination that will produce a HIGH at the output of a five-input AND gate?
2. What logic level should be applied to the second input of a two-input AND gate if the logic signal at the first input is to be inhibited (prevented) from reaching the output?
3. *True or false:* An AND gate output will always differ from an OR gate output for the same input conditions.

3-5 NOT OPERATION

The **NOT operation** is unlike the OR and AND operations because it can be performed on a single input variable. For example, if the variable A is subjected to the NOT operation, the result x can be expressed as

$$x = \overline{A}$$

where the overbar represents the NOT operation. This expression is read as "x equals NOT A" or "x equals the *inverse* of A" or "x equals the *complement* of A." Each of these is in common usage, and all indicate that the logic value of $x = \overline{A}$ *is opposite to* the logic value of A. The truth table in Figure 3-11(a) clarifies this for the two cases $A = 0$ and $A = 1$. That is,

$$0 = \overline{1} \quad \text{because 0 is not 1}$$

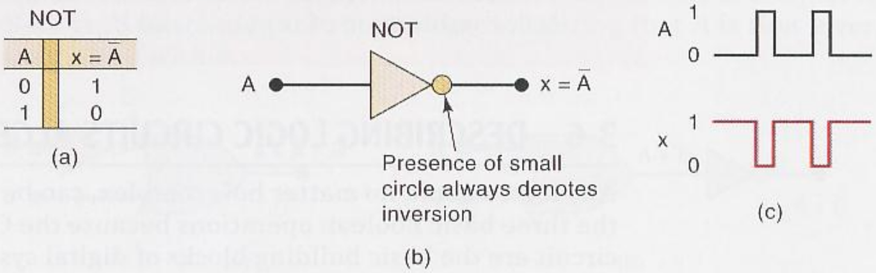
and

$$1 = \overline{0} \quad \text{because 1 is not 0}$$

The NOT operation is also referred to as **inversion** or **complementation**, and these terms will be used interchangeably throughout the book. Although we will always use the overbar indicator to represent inversion, it is important to mention that another indicator for inversion is the prime symbol ($'$). That is,

$$A' = \overline{A}$$

Both should be recognized as indicating the inversion operation.



A	$x = \overline{A}$
0	1
1	0

FIGURE 3-11 (a) Truth table; (b) symbol for the INVERTER (NOT circuit); (c) sample waveforms.

NOT Circuit (INVERTER)

Figure 3-11(b) shows the symbol for a **NOT circuit**, which is more commonly called an **INVERTER**. This circuit *always* has only a single input, and its output logic level is always opposite to the logic level of this input. Figure 3-11(c) shows how the INVERTER affects an input signal. It inverts (complements) the input signal at all points on the waveform so that whenever the input = 0, output = 1, and vice versa.

APPLICATION 3-1

Figure 3-12 shows a typical application of the NOT gate. The push button is wired to produce a logic 1 (true) when it is pressed. Sometimes we want to know if the push button is not being pressed, and so this circuit provides an expression that is true when the button is not pressed.

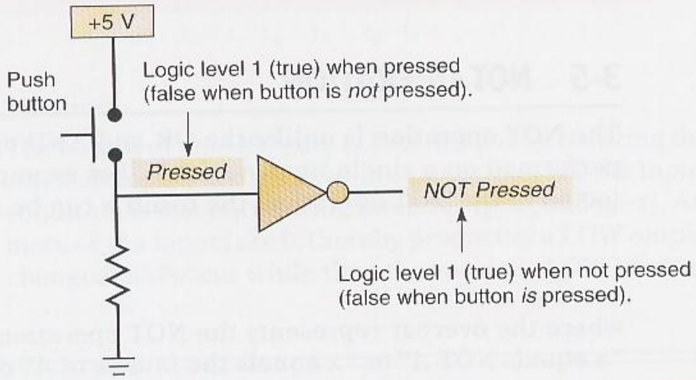

+5 V
Push button
Logic level 1 (true) when pressed
(false when button is *not* pressed).
Pressed
NOT Pressed
Logic level 1 (true) when not pressed
(false when button *is* pressed).

FIGURE 3-12 A NOT gate indicating a button is *not* pressed when its output is true.

Summary of Boolean Operations

The rules for the OR, AND, and NOT operations may be summarized as follows:

OR	*AND*	*NOT*
$0 + 0 = 0$	$0 \cdot 0 = 0$	$\overline{0} = 1$
$0 + 1 = 1$	$0 \cdot 1 = 0$	$\overline{1} = 0$
$1 + 0 = 1$	$1 \cdot 0 = 0$	
$1 + 1 = 1$	$1 \cdot 1 = 1$	

REVIEW QUESTIONS

1. The output of the INVERTER of Figure 3-11 is connected to the input of a second INVERTER. Determine the output level of the second INVERTER for each level of input *A*.
2. The output of the AND gate in Figure 3-7 is connected to the input of an INVERTER. Write the truth table showing the INVERTER output, *y*, for each combination of inputs *A* and *B*.

3-6 DESCRIBING LOGIC CIRCUITS ALGEBRAICALLY

Any logic circuit, no matter how complex, can be described completely using the three basic Boolean operations because the OR gate, AND gate, and NOT circuit are the basic building blocks of digital systems. For example, consider

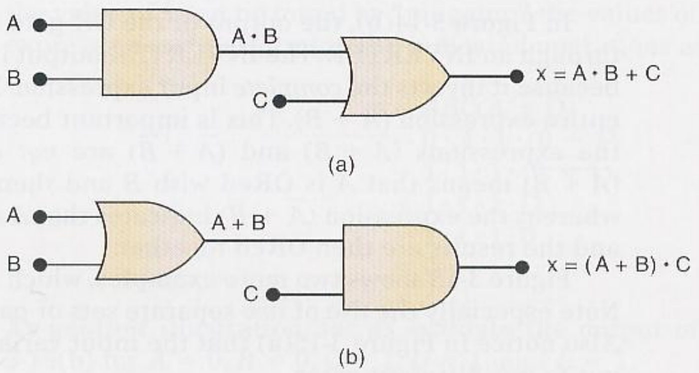


FIGURE 3-13 (a) Logic circuit with its Boolean expression; (b) logic circuit whose expression requires parentheses.

the circuit in Figure 3-13(a). This circuit has three inputs, A, B, and C, and a single output, x. Utilizing the Boolean expression for each gate, we can easily determine the expression for the output.

The expression for the AND gate output is written $A \cdot B$. This AND output is connected as an input to the OR gate along with C, another input. The OR gate operates on its inputs so that its output is the OR sum of the inputs. Thus, we can express the OR output as $x = A \cdot B + C$. (This final expression could also be written as $x = C + A \cdot B$ because it does not matter which term of the OR sum is written first.)

Operator Precedence

Occasionally, there may be confusion about which operation in an expression is performed first. The expression $A \cdot B + C$ can be interpreted in two different ways: (1) $A \cdot B$ is ORed with C, or (2) A is ANDed with the term $B + C$. To avoid this confusion, it will be understood that if an expression contains both AND and OR operations, the AND operations are performed first, unless there are *parentheses* in the expression, in which case the operation inside the parentheses is to be performed first. This is the same rule that is used in ordinary algebra to determine the order of operations.

To illustrate further, consider the circuit in Figure 3-13(b). The expression for the OR gate output is simply $A + B$. This output serves as an input to the AND gate along with another input, C. Thus, we express the output of the AND gate as $x = (A + B) \cdot C$. Note the use of parentheses here to indicate that A and B are ORed *first*, before their OR sum is ANDed with C. Without the parentheses it would be interpreted *incorrectly*, because $A + B \cdot C$ means that A is ORed with the product $B \cdot C$.

Circuits Containing INVERTERs

Whenever an INVERTER is present in a logic-circuit diagram, its output expression is simply equal to the input expression with a bar over it. Figure 3-14 shows two examples using INVERTERs. In Figure 3-14(a), input A is fed through an INVERTER, whose output is therefore $\overline{A}$. The INVERTER output is fed to an OR gate together with B, so that the *OR* output is equal to $\overline{A} + B$. Note that the bar is over the A alone, indicating that A is first inverted and then ORed with B.

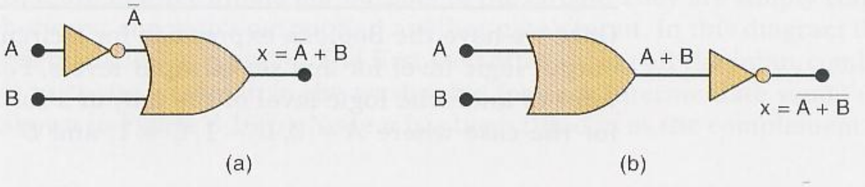


FIGURE 3-14 Circuits using INVERTERs.

In Figure 3-14(b), the output of the OR gate is equal to $A + B$ and is fed through an INVERTER. The INVERTER output is therefore equal to $\overline{(A + B)}$ because it inverts the *complete* input expression. Note that the bar covers the entire expression $(A + B)$. This is important because, as will be shown later, the expressions $\overline{(A + B)}$ and $(\overline{A} + \overline{B})$ are *not* equivalent. The expression $\overline{(A + B)}$ means that A is ORed with B and then their OR sum is inverted, whereas the expression $(\overline{A} + \overline{B})$ indicates that A is inverted and B is inverted and the results are then ORed together.

Figure 3-15 shows two more examples, which should be studied carefully. Note especially the use of *two* separate sets of parentheses in Figure 3-15(b). Also notice in Figure 3-15(a) that the input variable A is connected as an input to two different gates.

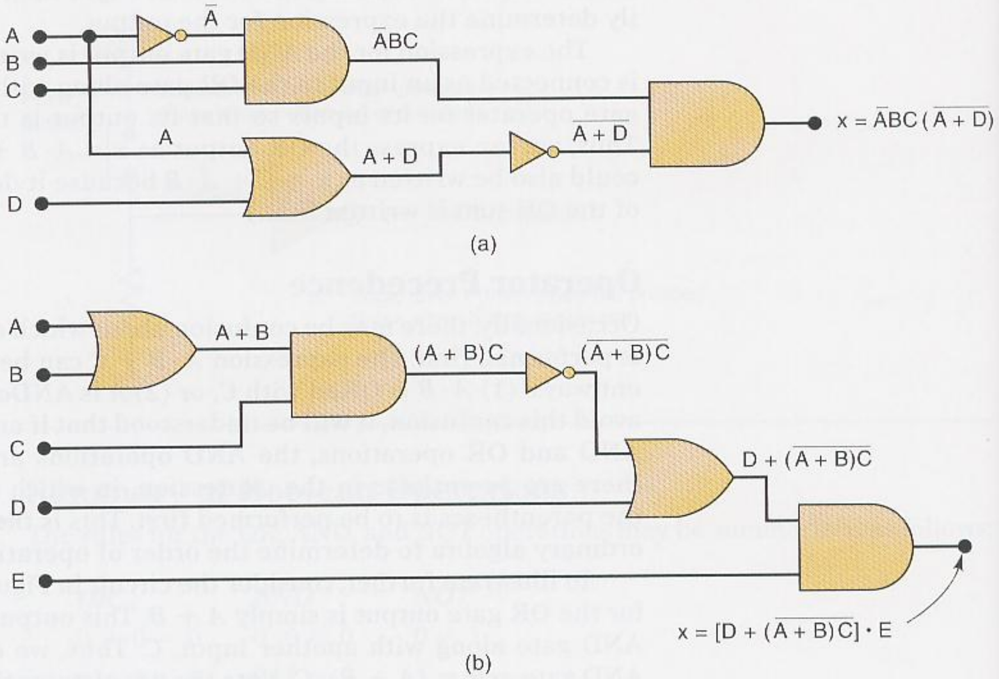

A
B
C
$\overline{A}$
$\overline{A}BC$
A
A + D
A + D
D
$x = \overline{A}BC\,\overline{(A + D)}$
(a)
A
B
A + B
(A + B)C
$\overline{(A + B)C}$
C
D
$D + \overline{(A + B)C}$
E
$x = [D + \overline{(A + B)C}] \cdot E$
(b)

FIGURE 3-15 More examples.

REVIEW QUESTIONS

1. In Figure 3-15(a), change each AND gate to an OR gate, and change the OR gate to an AND gate. Then write the expression for output x.
2. In Figure 3-15(b), change each AND gate to an OR gate, and each OR gate to an AND gate. Then write the expression for x.

3-7 EVALUATING LOGIC-CIRCUIT OUTPUTS

Once we have the Boolean expression for a circuit output, we can obtain the output logic level for any set of input levels. For example, suppose that we want to know the logic level of the output x for the circuit in Figure 3-15(a) for the case where $A = 0, B = 1, C = 1$, and $D = 1$. As in ordinary algebra, the value of x can be found by "plugging" the values of the variables into the expression and performing the indicated operations as follows:

$$\begin{aligned} x &= \overline{A}BC\overline{(A + D)} \\ &= \overline{0}\cdot 1\cdot 1\cdot \overline{(0 + 1)} \\ &= 1\cdot 1\cdot 1\cdot \overline{(0 + 1)} \\ &= 1\cdot 1\cdot 1\cdot (\overline{1}) \\ &= 1\cdot 1\cdot 1\cdot 0 \\ &= 0 \end{aligned}$$

As another illustration, let us evaluate the output of the circuit in Figure 3-15(b) for $A = 0, B = 0, C = 1, D = 1$, and $E = 1$.

$$\begin{aligned} x &= [D + \overline{(A + B)C}]\cdot E \\ &= [1 + \overline{(0 + 0)\cdot 1}]\cdot 1 \\ &= [1 + \overline{0\cdot 1}]\cdot 1 \\ &= [1 + \overline{0}]\cdot 1 \\ &= [1 + 1]\cdot 1 \\ &= 1\cdot 1 \\ &= 1 \end{aligned}$$

In general, the following rules must always be followed when evaluating a Boolean expression:

1. First, perform all inversions of single terms; that is, $\overline{0} = 1$ or $\overline{1} = 0$.
2. Then perform all operations within parentheses.
3. Perform an AND operation before an OR operation unless parentheses indicate otherwise.
4. If an expression has a bar over it, perform the operations inside the expression first and then invert the result.

For practice, determine the outputs of both circuits in Figure 3-15 for the case where all inputs are 1. The answers are $x = 0$ and $x = 1$, respectively.

Analysis Using a Table

Whenever you have a combinational logic circuit and you want to know how it works, the best way to analyze it is to use a truth table. The advantages of this method are:

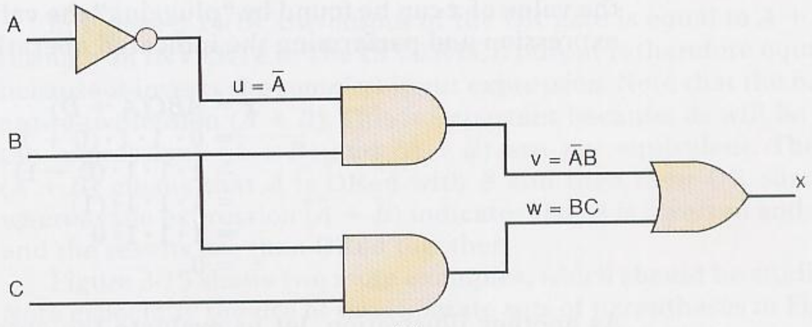
It allows you to analyze one gate or logic combination at a time.

It allows you to easily double-check your work.

When you are done, you have a table that is of tremendous benefit in troubleshooting the logic circuit.

Recall that a truth table lists all the possible input combinations in numerical order. For each possible input combination, we can determine the logic state at every point (node) in the logic circuit including the output. For example refer to Figure 3-16(a). There are several intermediate nodes in this circuit that are neither inputs nor outputs to the circuit. They are simply connections between one gate's output and another gate's input. In this diagram they have been labeled u, v, and w. The first step after listing all the input combinations is to create a column in the truth table for each intermediate signal (node) as shown in Figure 3-16(b). Node u has been filled in as the complement of A.

FIGURE 3-16 Analysis of a logic circuit using truth tables.



(a)

A	B	C	u= $\overline{A}$	v= $\overline{A}$B	w= BC	x= v+w
0	0	0	1			
0	0	1	1			
0	1	0	1			
0	1	1	1			
1	0	0	0			
1	0	1	0			
1	1	0	0			
1	1	1	0			

(b)

A	B	C	u= $\overline{A}$	v= $\overline{A}$B	w= BC	x= v+w
0	0	0	1	0		
0	0	1	1	0		
0	1	0	1	1		
0	1	1	1	1		
1	0	0	0	0		
1	0	1	0	0		
1	1	0	0	0		
1	1	1	0	0		

(c)

A	B	C	u= $\overline{A}$	v= $\overline{A}$B	w= BC	x= v+w
0	0	0	1	0	0	
0	0	1	1	0	0	
0	1	0	1	1	0	
0	1	1	1	1	1	
1	0	0	0	0	0	
1	0	1	0	0	0	
1	1	0	0	0	0	
1	1	1	0	0	1	

(d)

A	B	C	u= $\overline{A}$	v= $\overline{A}$B	w= BC	x= v+w
0	0	0	1	0	0	0
0	0	1	1	0	0	0
0	1	0	1	1	0	1
0	1	1	1	1	1	1
1	0	0	0	0	0	0
1	0	1	0	0	0	0
1	1	0	0	0	0	0
1	1	1	0	0	1	1

(e)

The next step is to fill in the values for column v as shown in Figure 3-16(c). From the diagram we can see that $v = \overline{A}B$. The node v should be HIGH when $\overline{A}$ (node u) is HIGH AND B is HIGH. This occurs whenever A is LOW AND B is HIGH. The third step is to predict the values at node w which is the logical product of BC. This column is HIGH whenever B is HIGH AND C is HIGH as shown in Figure 3-16(d). The final step is to logically combine columns v and w to predict the output x. Since $x = v + w$, the x output will be HIGH when v is HIGH OR w is HIGH as shown in Figure 3-16(e).

If you built this circuit and it was not producing the correct output for x under all conditions, this table could be used to find the trouble. The general procedure is to test the circuit under each combination of inputs. If any input combination produces an incorrect output (i.e., a fault), compare the actual logic state of each intermediate node in the circuit with the correct theoretical value in the table while applying that input condition. If the logic state for an intermediate node is *correct*, the problem must be farther to the right of that node. If the logic state for an intermediate node is *incorrect*, the problem must be to the left of that node (or that node is shorted to something). Detailed troubleshooting procedures and possible circuit faults will be covered more extensively in Chapter 4.

EXAMPLE 3-6

Analyze the operation of Figure 3-15(a) by creating a table showing the logic state at each node of the circuit.

Solution

Fill in the column for t by entering a 1 for all entries where $A = 0$ and $B = 1$ and $C = 1$.

Fill in the column for u by entering a 1 for all entries where $A = 1$ or $D = 1$.

Fill in the column for v by complementing all entries in column u.

Fill in the column for x by entering a 1 for all entries where $t = 1$ and $v = 1$.

A	B	C	D	$t = \overline{A}BC$	$u = A + D$	$v = \overline{A + D}$	$x = tv$
0	0	0	0	0	0	1	0
0	0	0	1	0	1	0	0
0	0	1	0	0	0	1	0
0	0	1	1	0	1	0	0
0	1	0	0	0	0	1	0
0	1	0	1	0	1	0	0
0	1	1	0	1	0	1	1
0	1	1	1	1	1	0	0
1	0	0	0	0	1	0	0
1	0	0	1	0	1	0	0
1	0	1	0	0	1	0	0
1	0	1	1	0	1	0	0
1	1	0	0	0	1	0	0
1	1	0	1	0	1	0	0
1	1	1	0	0	1	0	0
1	1	1	1	0	1	0	0

REVIEW QUESTIONS

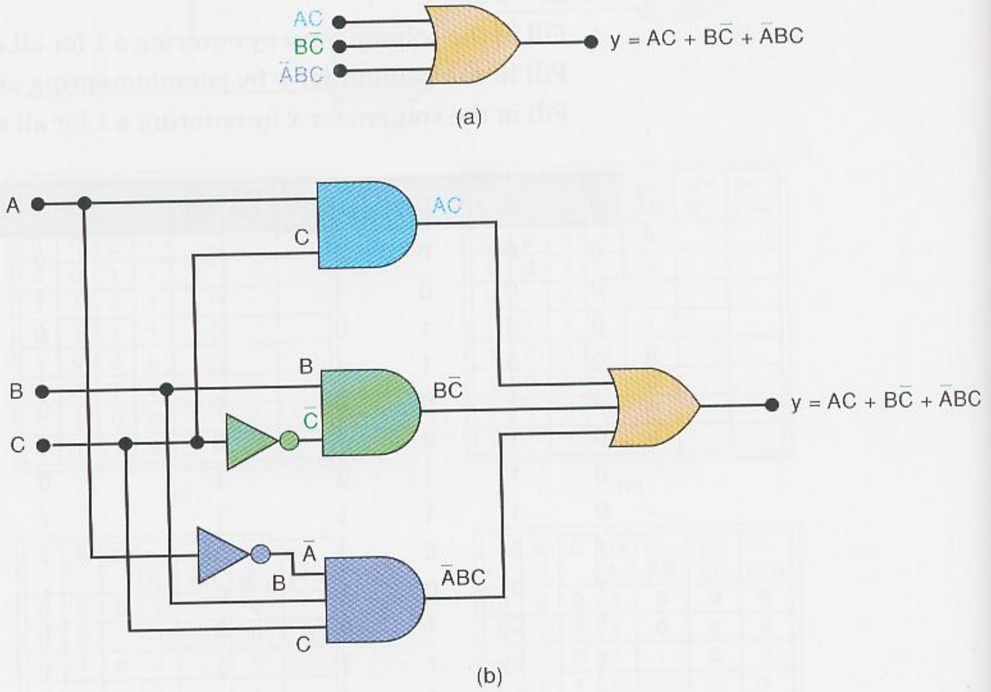
1. Use the expression for x to determine the output of the circuit in Figure 3-15(a) for the conditions $A = 0, B = 1, C = 1$, and $D = 0$.
2. Use the expression for x to determine the output of the circuit in Figure 3-15(b) for the conditions $A = B = E = 1, C = D = 0$.
3. Determine the answers to Questions 1 and 2 by finding the logic levels present at each gate output using a table as in Figure 3-16.

3-8 IMPLEMENTING CIRCUITS FROM BOOLEAN EXPRESSIONS

When the operation of a circuit is defined by a Boolean expression, we can draw a logic-circuit diagram directly from that expression. For example, if we needed a circuit that was defined by $x = A \cdot B \cdot C$, we would immediately know that all that was needed was a three-input AND gate. If we needed a circuit that was defined by $x = A + \overline{B}$, we would use a two-input OR gate with an INVERTER on one of the inputs. The same reasoning used for these simple cases can be extended to more complex circuits.

Suppose that we wanted to construct a circuit whose output is $y = AC + B\overline{C} + \overline{A}BC$. This Boolean expression contains three terms (AC, $B\overline{C}$, $\overline{A}BC$), which are ORed together. This tells us that a three-input OR gate is required with inputs that are equal to AC, $B\overline{C}$, and $\overline{A}BC$. This is illustrated in Figure 3-17(a), where a three-input OR gate is drawn with inputs labeled as AC, $B\overline{C}$, and $\overline{A}BC$.

FIGURE 3-17 Constructing a logic circuit from a Boolean expression.


AC
$B\overline{C}$
$\overline{A}BC$
$y = AC + B\overline{C} + \overline{A}BC$
(a)
A
C
AC
B
B
$\overline{C}$
$B\overline{C}$
C
$y = AC + B\overline{C} + \overline{A}BC$
$\overline{A}$
B
$\overline{A}BC$
C
(b)

Each OR gate input is an AND product term, which means that an AND gate with appropriate inputs can be used to generate each of these terms. This is shown in Figure 3-17(b), which is the final circuit diagram. Note the use of INVERTERs to produce the $\overline{A}$ and $\overline{C}$ terms required in the expression.

This same general approach can always be followed, although we shall find that there are some clever, more efficient techniques that can be employed. For now, however, this straightforward method will be used to minimize the number of new items that are to be learned.

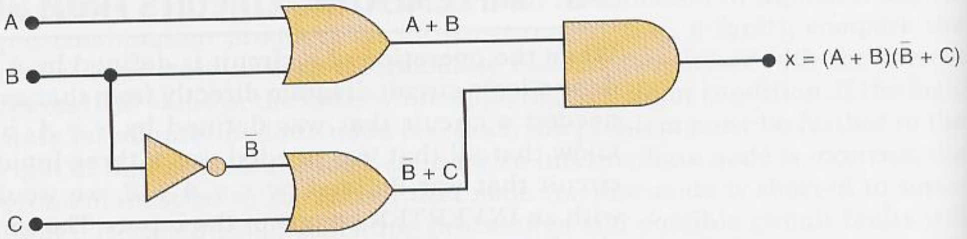
EXAMPLE 3-7

Draw the circuit diagram to implement the expression $x = (A + B)(\overline{B} + C)$.

Solution

This expression shows that the terms $A + B$ and $\overline{B} + C$ are inputs to an AND gate, and each of these two terms is generated from a separate OR gate. The result is drawn in Figure 3-18.

FIGURE 3-18 Example 3-7.


A
B
C
A + B
$\overline{B}$
$\overline{B} + C$
$x = (A + B)(\overline{B} + C)$

REVIEW QUESTIONS

1. Draw the circuit diagram that implements the expression $x = \overline{A}BC(\overline{A + D})$ using gates with no more than three inputs.
2. Draw the circuit diagram for the expression $y = AC + B\overline{C} + \overline{A}BC$.
3. Draw the circuit diagram for $x = [D + (\overline{A + B)C})] \cdot E$.

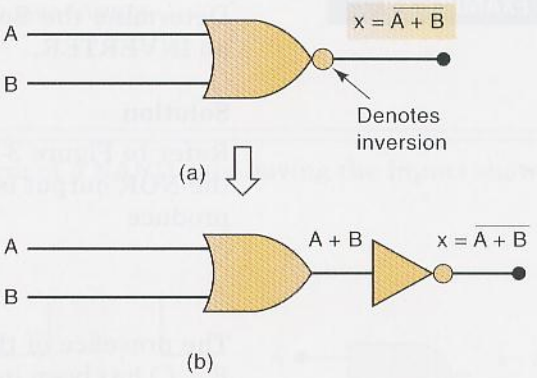
3-9 NOR GATES AND NAND GATES

Two other types of logic gates, NOR gates and NAND gates, are widely used in digital circuits. These gates actually combine the basic AND, OR, and NOT operations, so it is a relatively simple matter to write their Boolean expressions.

NOR Gate

The symbol for a two-input **NOR gate** is shown in Figure 3-19(a). It is the same as the OR gate symbol except that it has a small circle on the output. The small circle represents the inversion operation. Thus, the NOR gate operates like an OR gate followed by an INVERTER, so that the circuits in Figure 3-19(a) and (b) are equivalent, and the output expression for the NOR gate is $x = \overline{A + B}$.

FIGURE 3-19 (a) NOR symbol; (b) equivalent circuit; (c) truth table.



		OR	NOR
A	B	A + B	$\overline{A + B}$
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	0

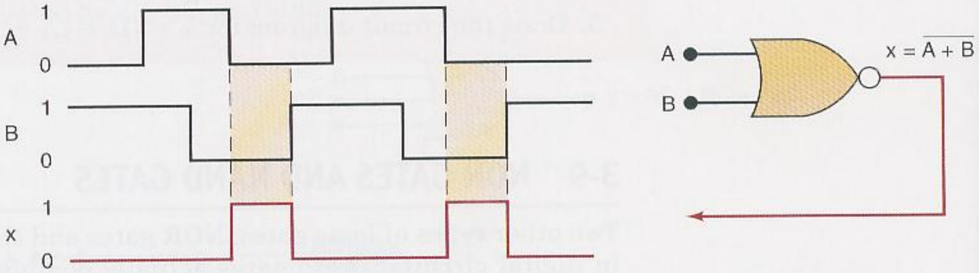
(c)

The truth table in Figure 3-19(c) shows that the NOR gate output is the exact inverse of the OR gate output for all possible input conditions. An OR gate output goes HIGH when any input is HIGH; the NOR gate output goes LOW when any input is HIGH. This same operation can be extended to NOR gates with more than two inputs.

EXAMPLE 3-8

Determine the waveform at the output of a NOR gate for the input waveforms shown in Figure 3-20.

FIGURE 3-20 Example 3-8.



Solution

One way to determine the NOR output waveform is to find first the OR output waveform and then invert it (change all 1s to 0s, and vice versa). Another way utilizes the fact that a NOR gate output will be HIGH *only* when all inputs are LOW. Thus, you can examine the input waveforms, find those time intervals where they are all LOW, and make the NOR output HIGH for those intervals. The NOR output will be LOW for all other time intervals. The resultant output waveform is shown in the figure.

EXAMPLE 3-9

Determine the Boolean expression for a three-input NOR gate followed by an INVERTER.

Solution

Refer to Figure 3-21, where the circuit diagram is shown. The expression at the NOR output is $(\overline{A + B + C})$, which is then fed through an INVERTER to produce

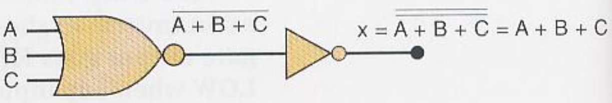
$$x = \overline{(\overline{A + B + C})}$$

The presence of the double inversion signs indicates that the quantity $(A + B + C)$ has been inverted and then inverted again. It should be clear that this simply results in the expression $(A + B + C)$ being unchanged. That is,

$$x = \overline{(\overline{A + B + C})} = (A + B + C)$$

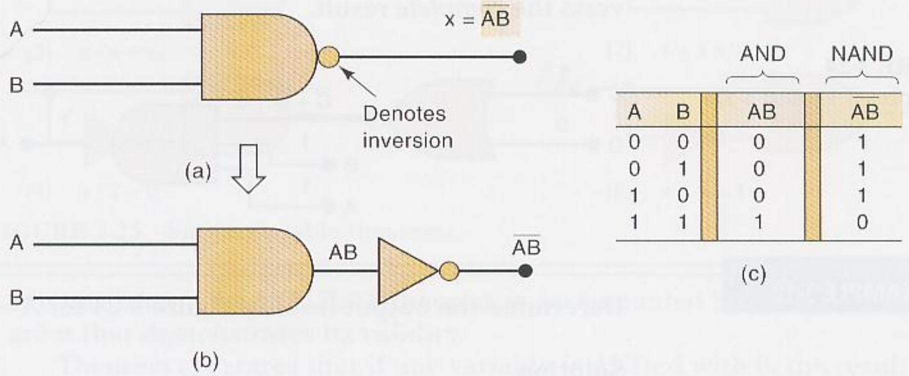
Whenever two inversion bars are over the same variable or quantity, they cancel each other out, as in the example above. However, in cases such as $\overline{A} + \overline{B}$ the inversion bars do not cancel. This is because the smaller inversion bars invert the single variables A and B, while the wide bar inverts the quantity $(\overline{A} + \overline{B})$. Thus, $\overline{\overline{A} + \overline{B}} \neq A + B$. Similarly, $\overline{\overline{A}\,\overline{B}} \neq AB$.

FIGURE 3-21 Example 3-9.



NAND Gate

The symbol for a two-input **NAND gate** is shown in Figure 3-22(a). It is the same as the AND gate symbol except for the small circle on its output. Once again, this small circle denotes the inversion operation. Thus, the NAND operates like an AND gate followed by an INVERTER, so that the circuits of Figure 3-22(a) and (b) are equivalent, and the output expression for the NAND gate is $x = \overline{AB}$.



A	B	AB	$\overline{AB}$
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0

FIGURE 3-22 (a) NAND symbol; (b) equivalent circuit; (c) truth table.

The truth table in Figure 3-22(c) shows that the NAND gate output is the exact inverse of the AND gate for all possible input conditions. The AND output goes HIGH only when all inputs are HIGH, while the NAND output goes LOW only when all inputs are HIGH. This same characteristic is true of NAND gates having more than two inputs.

EXAMPLE 3-10

Determine the output waveform of a NAND gate having the inputs shown in Figure 3-23.

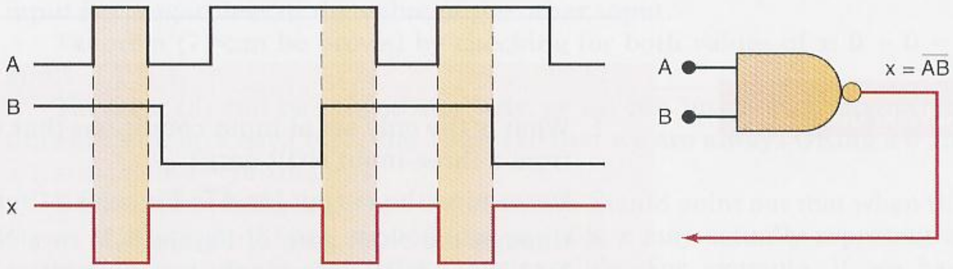


FIGURE 3-23 Example 3-10.

Solution

One way is to draw first the output waveform for an AND gate and then invert it. Another way utilizes the fact that a NAND output will be LOW only when all inputs are HIGH. Thus, you can find those time intervals during which the inputs are all HIGH, and make the NAND output LOW for those intervals. The output will be HIGH at all other times.

EXAMPLE 3-11

Implement the logic circuit that has the expression $x = \overline{AB \cdot \overline{(C + D)}}$ using only NOR and NAND gates.

Solution

The $\overline{(C + D)}$ term is the expression for the output of a NOR gate. This term is ANDed with A and B, and the result is inverted; this, of course, is the NAND operation. Thus, the circuit is implemented as shown in Figure 3-24. Note that the NAND gate first ANDs the A, B, and $\overline{(C + D)}$ terms, and then it inverts the *complete* result.

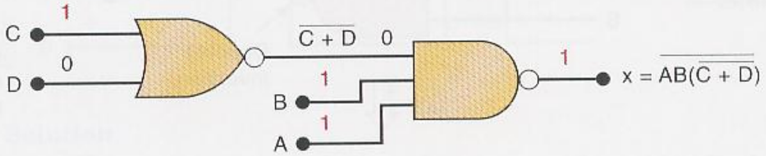


FIGURE 3-24 Examples 3-11 and 3-12.

EXAMPLE 3-12

Determine the output level in Figure 3-24 for $A = B = C = 1$ and $D = 0$.

Solution

In the first method we use the expression for x.

$$
\begin{aligned}
x &= \overline{AB\overline{(C + D)}} \\
&= \overline{1 \cdot 1 \cdot \overline{(1 + 0)}} \\
&= \overline{1 \cdot 1 \cdot \overline{(1)}} \\
&= \overline{1 \cdot 1 \cdot 0} \\
&= \overline{0} = 1
\end{aligned}
$$

In the second method, we write down the input logic levels on the circuit diagram (shown in color in Figure 3-24) and follow these levels through each gate to the final output. The NOR gate has inputs of 1 and 0 to produce an output of 0 (an OR would have produced an output of 1). The NAND gate thus has input levels of 0, 1, and 1 to produce an output of 1 (an AND would have produced an output of 0).

REVIEW QUESTIONS

1. What is the only set of input conditions that will produce a HIGH output from a three-input NOR gate?
2. Determine the output level in Figure 3-24 for $A = B = 1$, $C = D = 0$.
3. Change the NOR gate of Figure 3-24 to a NAND gate, and change the NAND to a NOR. What is the new expression for x?

3-10 BOOLEAN THEOREMS

We have seen how Boolean algebra can be used to help analyze a logic circuit and express its operation mathematically. We will continue our study of Boolean algebra by investigating the various **Boolean theorems** (rules) that can help us to simplify logic expressions and logic circuits. The first group of theorems is given in Figure 3-25. In each theorem, x is a logic variable that

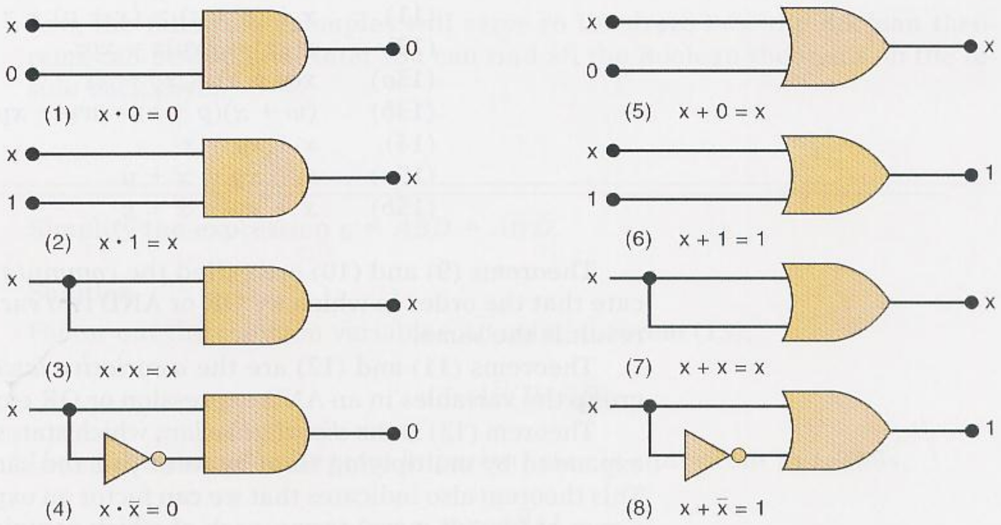


FIGURE 3-25 Single-variable theorems.

can be either a 0 or a 1. Each theorem is accompanied by a logic-circuit diagram that demonstrates its validity.

Theorem (1) states that if any variable is ANDed with 0, the result must be 0. This is easy to remember because the AND operation is just like ordinary multiplication, where we know that anything multiplied by 0 is 0. We also know that the output of an AND gate will be 0 whenever any input is 0, regardless of the level on the other input.

Theorem (2) is also obvious by comparison with ordinary multiplication.

Theorem (3) can be proved by trying each case. If $x = 0$, then $0 \cdot 0 = 0$; if $x = 1$, then $1 \cdot 1 = 1$. Thus, $x \cdot x = x$.

Theorem (4) can be proved in the same manner. However, it can also be reasoned that at any time either x or its inverse $\overline{x}$ must be at the 0 level, and so their AND product always must be 0.

Theorem (5) is straightforward, since 0 *added* to anything does not affect its value, either in regular addition or in OR addition.

Theorem (6) states that if any variable is ORed with 1, the result will always be 1. We check this for both values of x: $0 + 1 = 1$ and $1 + 1 = 1$. Equivalently, we can remember that an OR gate output will be 1 when *any* input is 1, regardless of the value of the other input.

Theorem (7) can be proved by checking for both values of x: $0 + 0 = 0$ and $1 + 1 = 1$.

Theorem (8) can be proved similarly, or we can just reason that at any time either x or $\overline{x}$ must be at the 1 level so that we are always ORing a 0 and a 1, which always results in 1.

Before introducing any more theorems, we should point out that when theorems (1) through (8) are applied, the variable x may actually represent an expression containing more than one variable. For example, if we have $A\overline{B}(\overline{A\overline{B}})$, we can invoke theorem (4) by letting $x = A\overline{B}$. Thus, we can say that $A\overline{B}(\overline{A\overline{B}}) = 0$. The same idea can be applied to the use of any of these theorems.

Multivariable Theorems

The theorems presented below involve more than one variable:

(9) $x + y = y + x$
(10) $x \cdot y = y \cdot x$

(11) $x + (y + z) = (x + y) + z = x + y + z$
(12) $x(yz) = (xy)z = xyz$
(13*a*) $x(y + z) = xy + xz$
(13*b*) $(w + x)(y + z) = wy + xy + wz + xz$
(14) $x + xy = x$
(15*a*) $x + \overline{x}y = x + y$
(15*b*) $\overline{x} + xy = \overline{x} + y$

Theorems (9) and (10) are called the *commutative laws.* These laws indicate that the order in which we OR or AND two variables is unimportant; the result is the same.

Theorems (11) and (12) are the *associative laws,* which state that we can group the variables in an AND expression or OR expression any way we want.

Theorem (13) is the *distributive law,* which states that an expression can be expanded by multiplying term by term just the same as in ordinary algebra. This theorem also indicates that we can factor an expression. That is, if we have a sum of two (or more) terms, each of which contains a common variable, the common variable can be factored out just as in ordinary algebra. For example, if we have the expression $A\overline{B}C + \overline{A}\,\overline{B}\,\overline{C}$, we can factor out the $\overline{B}$ variable:

$$A\overline{B}C + \overline{A}\,\overline{B}\,\overline{C} = \overline{B}(AC + \overline{A}\,\overline{C})$$

As another example, consider the expression $ABC + ABD$. Here the two terms have the variables A and B in common, and so $A \cdot B$ can be factored out of both terms. That is,

$$ABC + ABD = AB(C + D)$$

Theorems (9) to (13) are easy to remember and use because they are identical to those of ordinary algebra. Theorems (14) and (15), on the other hand, do not have any counterparts in ordinary algebra. Each can be proved by trying all possible cases for x and y. This is illustrated (for theorem 14) by creating an analysis table for the equation $x + xy$ as follows:

x	y	xy	x + xy
0	0	0	0
0	1	0	0
1	0	0	1
1	1	1	1

Notice that the value of the entire expression $(x + xy)$ is always the same as x.

Theorem (14) can also be proved by factoring and using theorems (6) and (2) as follows:

$$
\begin{aligned}
x + xy &= x(1 + y) \\
&= x \cdot 1 && \text{[using theorem (6)]} \\
&= x && \text{[using theorem (2)]}
\end{aligned}
$$

All of these Boolean theorems can be useful in simplifying a logic expression—that is, in reducing the number of terms in the expression. When this is done, the reduced expression will produce a circuit that is less complex than the one that the original expression would have produced. A good portion of the next chapter will be devoted to the process of circuit simplification. For now, the following examples will serve to illustrate how the Boolean theorems can be applied. **Note:** You can find all the Boolean theorems on the inside back cover.

EXAMPLE 3-13

Simplify the expression $y = A\overline{B}D + A\overline{B}\,\overline{D}$.

Solution

Factor out the common variables $A\overline{B}$ using theorem (13):

$$y = A\overline{B}(D + \overline{D})$$

Using theorem (8), the term in parentheses is equivalent to 1. Thus,

$$\begin{aligned} y &= A\overline{B} \cdot 1 \\ &= A\overline{B} \qquad \text{[using theorem (2)]} \end{aligned}$$

EXAMPLE 3-14

Simplify $z = (\overline{A} + B)(A + B)$.

Solution

The expression can be expanded by multiplying out the terms [theorem (13)]:

$$z = \overline{A} \cdot A + \overline{A} \cdot B + B \cdot A + B \cdot B$$

Invoking theorem (4), the term $\overline{A} \cdot A = 0$. Also, $B \cdot B = B$ [theorem (3)]:

$$z = 0 + \overline{A} \cdot B + B \cdot A + B = \overline{A}B + AB + B$$

Factoring out the variable B [theorem (13)], we have

$$z = B(\overline{A} + A + 1)$$

Finally, using theorems (2) and (6),

$$z = B$$

EXAMPLE 3-15

Simplify $x = ACD + \overline{A}BCD$.

Solution

Factoring out the common variables CD, we have

$$x = CD(A + \overline{A}B)$$

Utilizing theorem (15a), we can replace $A + \overline{A}B$ by $A + B$, so

$$\begin{aligned} x &= CD(A + B) \\ &= ACD + BCD \end{aligned}$$

REVIEW QUESTIONS

1. Use theorems (13) and (14) to simplify $y = A\overline{C} + AB\overline{C}$.
2. Use theorems (13) and (8) to simplify $y = \overline{A}\,\overline{B}C\overline{D} + \overline{A}\,\overline{B}\,\overline{C}\,\overline{D}$.
3. Use theorems (13) and (15b) to simplify $y = \overline{A}D + ABD$.

3-11 DEMORGAN'S THEOREMS

Two of the most important theorems of Boolean algebra were contributed by a great mathematician named DeMorgan. **DeMorgan's theorems** are extremely useful in simplifying expressions in which a product or sum of variables is inverted. The two theorems are:

$$(16)\quad (\overline{x + y}) = \overline{x} \cdot \overline{y}$$
$$(17)\quad (\overline{x \cdot y}) = \overline{x} + \overline{y}$$

Theorem (16) says that when the OR sum of two variables is inverted, this is the same as inverting each variable individually and then ANDing these inverted variables. Theorem (17) says that when the AND product of two variables is inverted, this is the same as inverting each variable individually and then ORing them. Each of DeMorgan's theorems can readily be proven by checking for all possible combinations of x and y. This will be left as an end-of-chapter exercise.

Although these theorems have been stated in terms of single variables x and y, they are equally valid for situations where x and/or y are expressions that contain more than one variable. For example, let's apply them to the expression $(\overline{A\overline{B} + C})$ as shown below:

$$(\overline{A\overline{B} + C}) = (\overline{A\overline{B}}) \cdot \overline{C}$$

Note that we used theorem (16) and treated $A\overline{B}$ as x and C as y. The result can be further simplified because we have a product $A\overline{B}$ that is inverted. Using theorem (17), the expression becomes

$$\overline{A\overline{B}} \cdot \overline{C} = (\overline{A} + \overline{\overline{B}}) \cdot \overline{C}$$

Notice that we can replace $\overline{\overline{B}}$ by B, so that we finally have

$$(\overline{A} + B) \cdot \overline{C} = \overline{A}\,\overline{C} + B\overline{C}$$

This final result contains only inverter signs that invert a single variable.

EXAMPLE 3-16

Simplify the expression $z = \overline{(\overline{A} + C) \cdot (B + \overline{D})}$ to one having only single variables inverted.

Solution

Using theorem (17), and treating $(\overline{A} + C)$ as x and $(B + \overline{D})$ as y, we have

$$z = (\overline{\overline{A} + C}) + (\overline{B + \overline{D}})$$

We can think of this as breaking the large inverter sign down the middle and changing the AND sign (·) to an OR sign (+). Now the term $\overline{(\overline{A} + C)}$ can be simplified by applying theorem (16). Likewise, $\overline{(B + \overline{D})}$ can be simplified:

$$\begin{aligned} z &= \overline{(\overline{A} + C)} + \overline{(B + \overline{D})} \\ &= (\overline{\overline{A}} \cdot \overline{C}) + \overline{B} \cdot \overline{\overline{D}} \end{aligned}$$

Here we have broken the larger inverter signs down the middle and replaced the (+) with a (·). Canceling out the double inversions, we have finally

$$z = A\overline{C} + \overline{B}D$$

Example 3-16 points out that when using DeMorgan's theorems to reduce an expression, we may break an inverter sign at any point in the expression and change the operator sign at that point in the expression to its opposite (+ is changed to ·, and vice versa). This procedure is continued until the expression is reduced to one in which only single variables are inverted. Two more examples are given below.

Example 1

$$\begin{aligned} z &= \overline{A + \overline{B} \cdot C} \\ &= \overline{A} \cdot (\overline{\overline{B} \cdot C}) \\ &= \overline{A} \cdot (\overline{\overline{B}} + \overline{C}) \\ &= \overline{A} \cdot (B + \overline{C}) \end{aligned}$$

Example 2

$$\begin{aligned} \omega &= \overline{(A + BC) \cdot (D + EF)} \\ &= (\overline{A + BC}) + (\overline{D + EF}) \\ &= (\overline{A} \cdot \overline{BC}) + (\overline{D} \cdot \overline{EF}) \\ &= [\overline{A} \cdot (\overline{B} + \overline{C})] + [\overline{D} \cdot (\overline{E} + \overline{F})] \\ &= \overline{A}\,\overline{B} + \overline{A}\,\overline{C} + \overline{D}\,\overline{E} + \overline{D}\,\overline{F} \end{aligned}$$

DeMorgan's theorems are easily extended to more than two variables. For example, it can be proved that

$$\begin{aligned} \overline{x + y + z} &= \overline{x} \cdot \overline{y} \cdot \overline{z} \\ \overline{x \cdot y \cdot z} &= \overline{x} + \overline{y} + \overline{z} \end{aligned}$$

Here, we see that the large inverter sign is broken at *two* points in the expression and the operator sign is changed to its opposite. This can be extended to any number of variables. Again, realize that the variables can themselves be expressions rather than single variables. Here is another example.

$$\begin{aligned} x &= \overline{\overline{AB} \cdot \overline{CD} \cdot \overline{EF}} \\ &= \overline{\overline{AB}} + \overline{\overline{CD}} + \overline{\overline{EF}} \\ &= AB + CD + EF \end{aligned}$$

Implications of DeMorgan's Theorems

Let us examine theorems (16) and (17) from the standpoint of logic circuits. First, consider theorem (16):

$$\overline{x + y} = \overline{x} \cdot \overline{y}$$

The left-hand side of the equation can be viewed as the output of a NOR gate whose inputs are x and y. The right-hand side of the equation, on the other

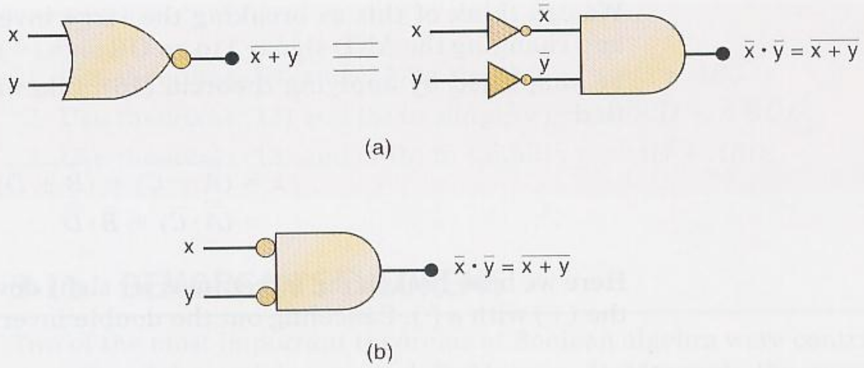

x
y
$\overline{x+y}$
x
y
$\overline{x}$
$\overline{y}$
$\overline{x} \cdot \overline{y} = \overline{x+y}$
(a)
x
y
$\overline{x} \cdot \overline{y} = \overline{x+y}$
(b)

FIGURE 3-26 (a) Equivalent circuits implied by theorem (16); (b) alternative symbol for the NOR function.

hand, is the result of first inverting both x and y and then putting them through an AND gate. These two representations are equivalent and are illustrated in Figure 3-26(a). What this means is that an AND gate with INVERTERs on each of its inputs is equivalent to a NOR gate. In fact, both representations are used to represent the NOR function. When the AND gate with inverted inputs is used to represent the NOR function, it is usually drawn as shown in Figure 3-26(b), where the small circles on the inputs represent the inversion operation.

Now consider theorem (17):

$$\overline{x \cdot y} = \overline{x} + \overline{y}$$

The left side of the equation can be implemented by a NAND gate with inputs x and y. The right side can be implemented by first inverting inputs x and y and then putting them through an OR gate. These two equivalent representations are shown in Figure 3-27(a). The OR gate with INVERTERs on each of its inputs is equivalent to the NAND gate. In fact, both representations are used to represent the NAND function. When the OR gate with inverted inputs is used to represent the NAND function, it is usually drawn as shown in Figure 3-27(b), where the circles again represent inversion.

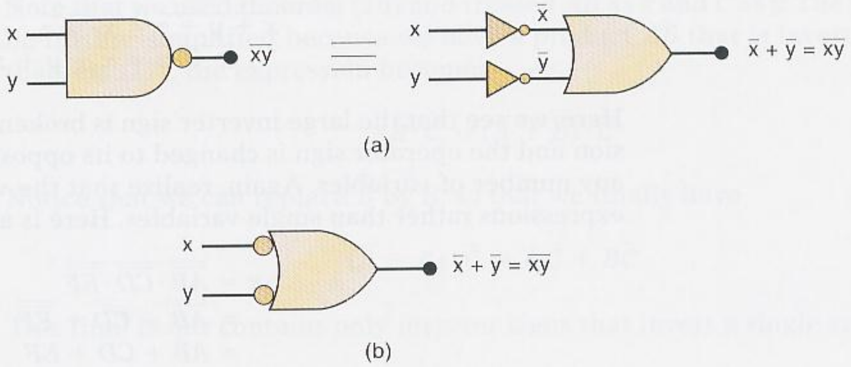

x
y
$\overline{xy}$
x
y
$\overline{x}$
$\overline{y}$
$\overline{x} + \overline{y} = \overline{xy}$
(a)
x
y
$\overline{x} + \overline{y} = \overline{xy}$
(b)

FIGURE 3-27 (a) Equivalent circuits implied by theorem (17); (b) alternative symbol for the NAND function.

EXAMPLE 3-17

Determine the output expression for the circuit of Figure 3-28 and simplify it using DeMorgan's theorems.

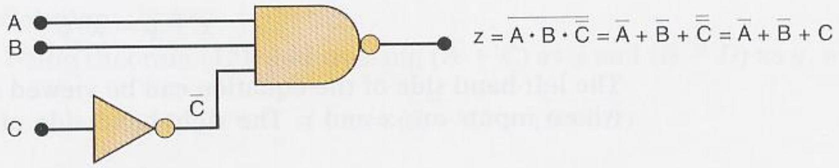

A
B
C
$\overline{C}$
$z = \overline{A \cdot B \cdot \overline{C}} = \overline{A} + \overline{B} + \overline{\overline{C}} = \overline{A} + \overline{B} + C$

FIGURE 3-28 Example 3-17.

Solution

The expression for z is $z = \overline{AB\overline{C}}$. Use DeMorgan's theorem to break the large inversion sign:

$$z = \overline{A} + \overline{B} + \overline{\overline{C}}$$

Cancel the double inversions over C to obtain

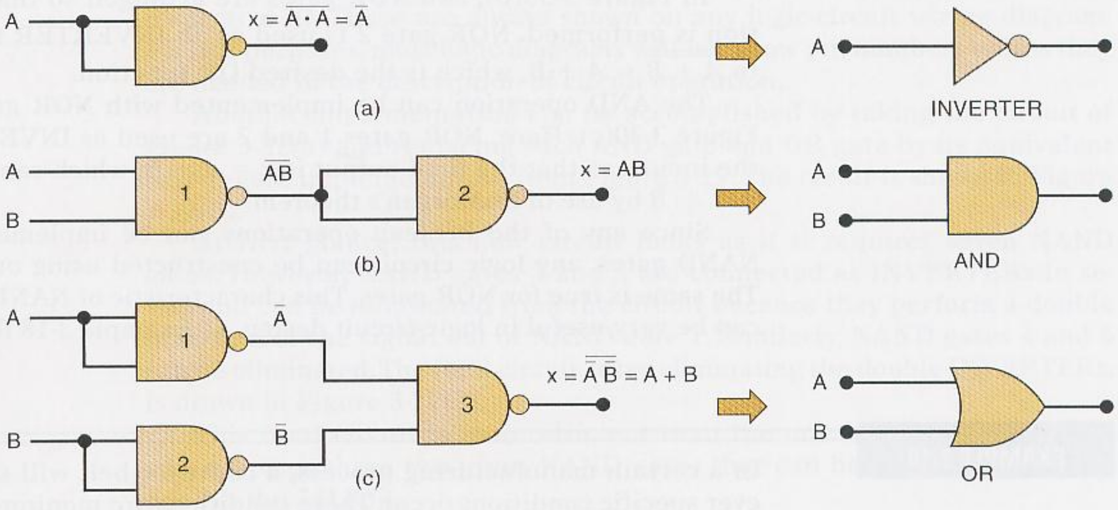
$$z = \overline{A} + \overline{B} + C$$

REVIEW QUESTIONS

1. Use DeMorgan's theorems to convert the expression $z = \overline{(A + B)\cdot\overline{C}}$ to one that has only single-variable inversions.
2. Repeat question 1 for the expression $y = \overline{R\overline{S}T + \overline{Q}}$.
3. Implement a circuit having output expression $z = \overline{A}\,\overline{B}C$ using only a NOR gate and an INVERTER.
4. Use DeMorgan's theorems to convert $y = \overline{A + \overline{B} + \overline{C}D}$ to an expression containing only single-variable inversions.

3-12 UNIVERSALITY OF NAND GATES AND NOR GATES

All Boolean expressions consist of various combinations of the basic operations of OR, AND, and INVERT. Therefore, any expression can be implemented using combinations of OR gates, AND gates, and INVERTERs. It is possible, however, to implement any logic expression using *only* NAND gates and no other type of gate. This is because NAND gates, in the proper combination, can be used to perform each of the Boolean operations OR, AND, and INVERT. This is demonstrated in Figure 3-29.



A
x = $\overline{A \cdot A}$ = $\overline{A}$
(a)
A
INVERTER
A
B
1
$\overline{AB}$
2
x = AB
(b)
A
B
AND
A
1
$\overline{A}$
3
x = $\overline{\overline{A}\,\overline{B}}$ = A + B
B
2
$\overline{B}$
(c)
A
B
OR

FIGURE 3-29 NAND gates can be used to implement any Boolean function.

First, in Figure 3-29(a), we have a two-input NAND gate whose inputs are purposely connected together so that the variable A is applied to both. In this configuration, the NAND simply acts as INVERTER because its output is $x = \overline{A \cdot A} = \overline{A}$.

In Figure 3-29(b), we have two NAND gates connected so that the AND operation is performed. NAND gate 2 is used as an INVERTER to change $\overline{AB}$ to $\overline{\overline{AB}} = AB$, which is the desired AND function.

The OR operation can be implemented using NAND gates connected as shown in Figure 3-29(c). Here NAND gates 1 and 2 are used as INVERTERs to invert the inputs, so that the final output is $x = \overline{\overline{A} \cdot \overline{B}}$, which can be simplified to $x = A + B$ using DeMorgan's theorem.

In a similar manner, it can be shown that NOR gates can be arranged to implement any of the Boolean operations. This is illustrated in Figure 3-30. Part (a) shows that a NOR gate with its inputs connected together behaves as an INVERTER because the output is $x = \overline{A + A} = \overline{A}$.

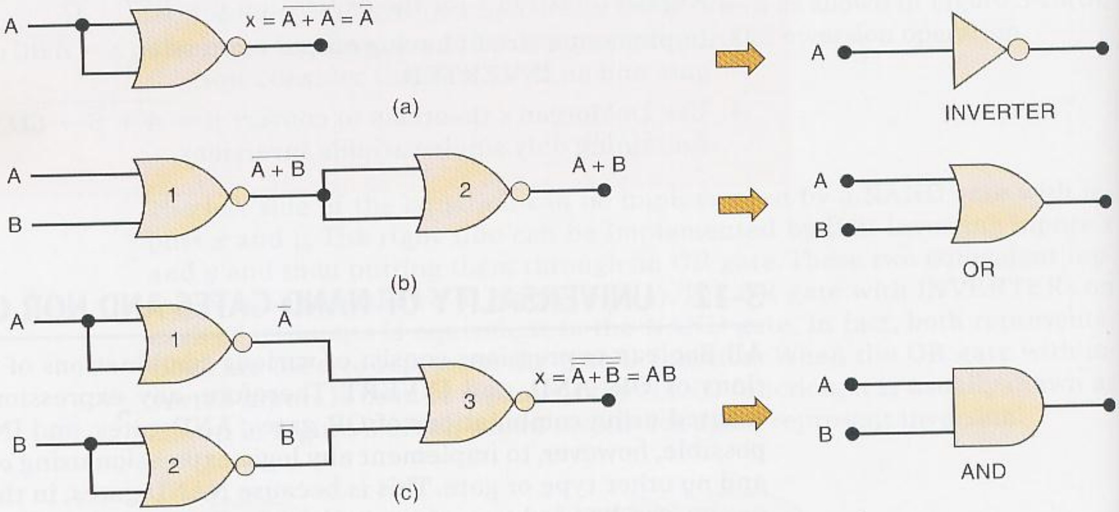


FIGURE 3-30 NOR gates can be used to implement any Boolean operation.

In Figure 3-30(b), two NOR gates are arranged so that the OR operation is performed. NOR gate 2 is used as an INVERTER to change $\overline{A + B}$ to $\overline{\overline{A + B}} = A + B$, which is the desired OR function.

The AND operation can be implemented with NOR gates as shown in Figure 3-30(c). Here, NOR gates 1 and 2 are used as INVERTERs to invert the inputs, so that the final output is $x = \overline{\overline{A} + \overline{B}}$, which can be simplified to $x = A \cdot B$ by use of DeMorgan's theorem.

Since any of the Boolean operations can be implemented using only NAND gates, any logic circuit can be constructed using only NAND gates. The same is true for NOR gates. This characteristic of NAND and NOR gates can be very useful in logic-circuit design, as Example 3-18 illustrates.

EXAMPLE 3-18

In a certain manufacturing process, a conveyor belt will shut down whenever specific conditions occur. These conditions are monitored and reflected by the states of four logic signals as follows: signal A will be HIGH whenever the conveyor belt speed is too fast; signal B will be HIGH whenever the collection bin at the end of the belt is full; signal C will be HIGH when the belt tension is too high; signal D will be HIGH when the manual override is off.

A logic circuit is needed to generate a signal x that will go HIGH whenever conditions A and B exist simultaneously or whenever conditions C and D exist simultaneously. Clearly, the logic expression for x will be $x = AB + CD$. The circuit is to be implemented with a minimum number of ICs. The TTL integrated circuits shown in Figure 3-31 are available. Each IC is a *quad,* which means that it contains *four* identical gates on one chip.

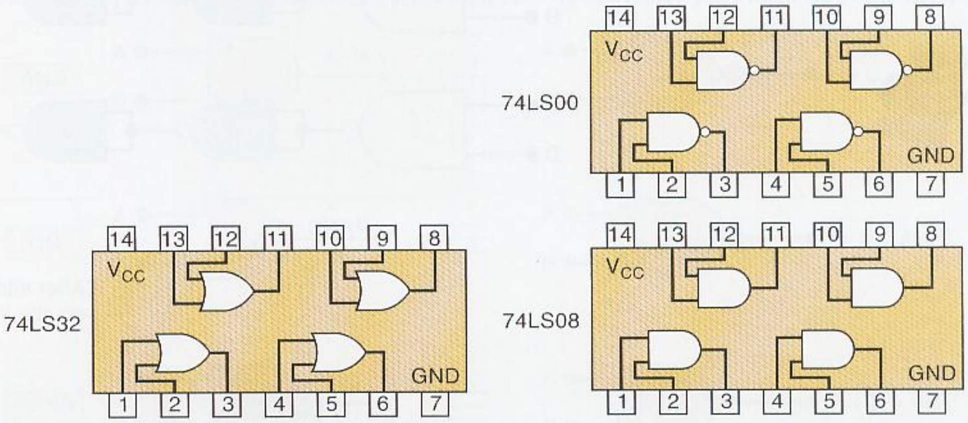


FIGURE 3-31 ICs available for Example 3-18.

Solution

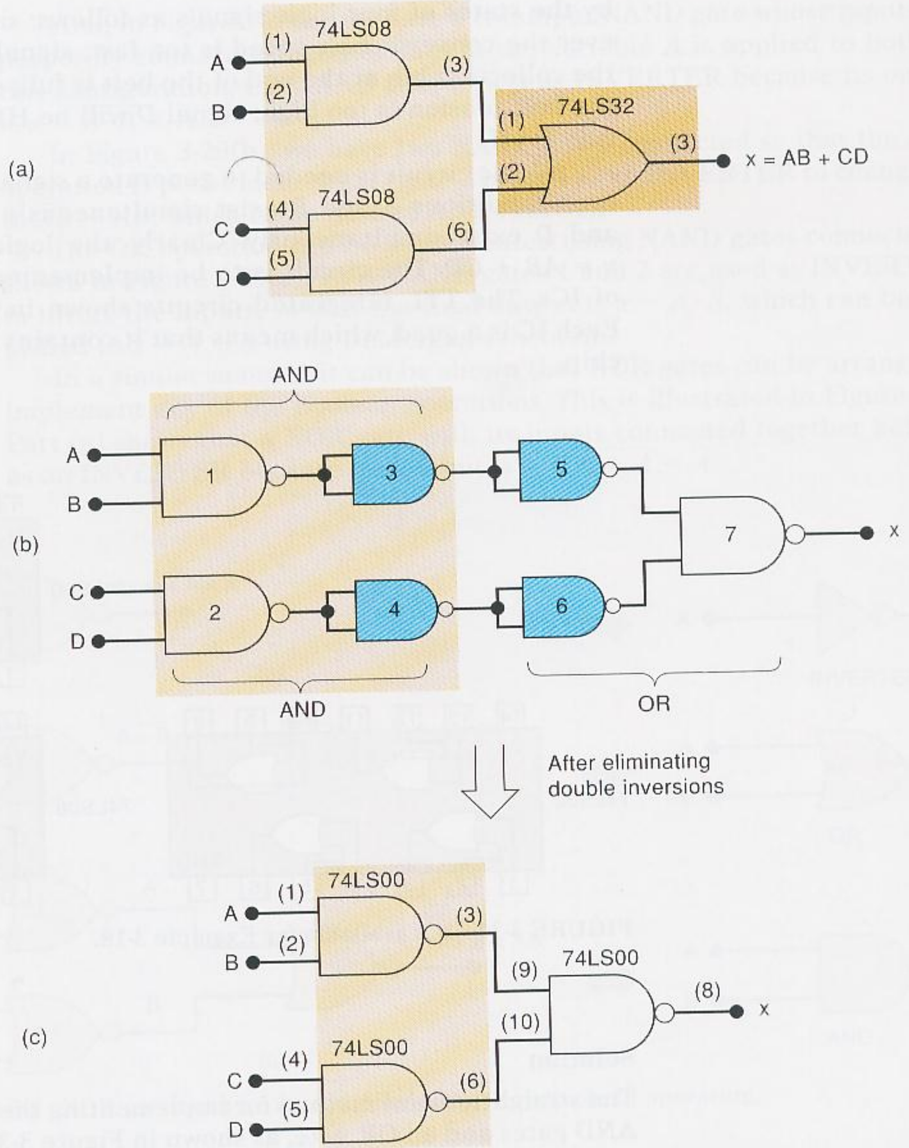
The straightforward method for implementing the given expression uses two AND gates and an OR gate, as shown in Figure 3-32(a). This implementation uses two gates from the 74LS08 IC and a single gate from the 74LS32 IC. The numbers in parentheses at each input and output are the pin numbers of the respective IC. These are always shown on any logic-circuit wiring diagram. For our purposes, most logic diagrams will not show pin numbers unless they are needed in the description of circuit operation.

Another implementation can be accomplished by taking the circuit of Figure 3-32(a) and replacing each AND gate and OR gate by its equivalent NAND gate implementation from Figure 3-29. The result is shown in Figure 3-32(b).

At first glance, this new circuit looks as if it requires seven NAND gates. However, NAND gates 3 and 5 are connected as INVERTERs in series and can be eliminated from the circuit because they perform a double inversion of the signal out of NAND gate 1. Similarly, NAND gates 4 and 6 can be eliminated. The final circuit, after eliminating the double INVERTERs, is drawn in Figure 3-32(c).

This final circuit is more efficient than the one in Figure 3-32(a) because it uses three two-input NAND gates that can be implemented from one IC, the 74LS00.

FIGURE 3-32 Possible implementations for Example 3-18.



(a)
74LS08
A
B
(1)
(2)
(3)
74LS32
(1)
(2)
(3)
x = AB + CD
74LS08
C
D
(4)
(5)
(6)
(b)
AND
A
B
C
D
1
2
3
4
5
6
7
x
AND
OR
After eliminating double inversions
(c)
74LS00
A
B
(1)
(2)
(3)
74LS00
(9)
(10)
(8)
x
74LS00
C
D
(4)
(5)
(6)

REVIEW QUESTIONS

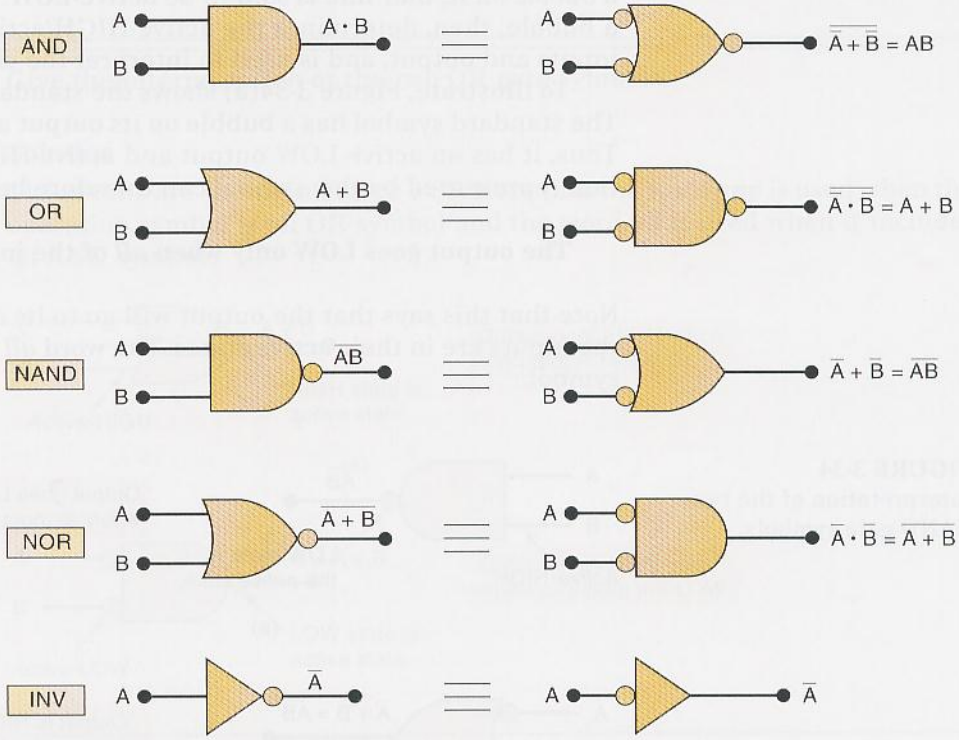
1. How many different ways do we now have to implement the inversion operation in a logic circuit?
2. Implement the expression $x = (A + B)(C + D)$ using OR and AND gates. Then implement the expression using only NOR gates by converting each OR and AND gate to its NOR implementation from Figure 3-30. Which circuit is more efficient?
3. Write the output expression for the circuit of Figure 3-32(c), and use DeMorgan's theorems to show that it is equivalent to the expression for the circuit of Figure 3-32(a).

3-13 ALTERNATE LOGIC-GATE REPRESENTATIONS

We have introduced the five basic logic gates (AND, OR, INVERTER, NAND, and NOR) and the standard symbols used to represent them on logic-circuit diagrams. Although you may find that some circuit diagrams still use these standard symbols exclusively, it has become increasingly more common to find circuit diagrams that utilize **alternate logic symbols** *in addition* to the standard symbols.

Before discussing the reasons for using an alternate symbol for a logic gate, we will present the alternate symbols for each gate and show that they are equivalent to the standard symbols. Refer to Figure 3-33; the left side of the illustration shows the standard symbol for each logic gate, and the right side shows the alternate symbol. The alternate symbol for each gate is obtained from the standard symbol by doing the following:

1. Invert each input and output of the standard symbol. This is done by adding bubbles (small circles) on input and output lines that do not have bubbles and by removing bubbles that are already there.
2. Change the operation symbol from AND to OR, or from OR to AND. (In the special case of the INVERTER, the operation symbol is not changed.)



AND
A
B
$A \cdot B$
≡
A
B
$\overline{\bar{A} + \bar{B}} = AB$
OR
A
B
$A + B$
≡
A
B
$\overline{\bar{A} \cdot \bar{B}} = A + B$
NAND
A
B
$\overline{AB}$
≡
A
B
$\bar{A} + \bar{B} = \overline{AB}$
NOR
A
B
$\overline{A + B}$
≡
A
B
$\bar{A} \cdot \bar{B} = \overline{A + B}$
INV
A
$\bar{A}$
≡
A
$\bar{A}$

FIGURE 3-33 Standard and alternate symbols for various logic gates and inverter.

For example, the standard NAND symbol is an AND symbol with a bubble on its output. Following the steps outlined above, remove the bubble from the output, and add a bubble to each input. Then change the AND symbol to an OR symbol. The result is an OR symbol with bubbles on its inputs.

We can easily prove that this alternate symbol is equivalent to the standard symbol by using DeMorgan's theorems and recalling that the bubble represents an inversion operation. The output expression from the standard NAND symbol is $\overline{AB} = \bar{A} + \bar{B}$, which is the same as the output expression for the alternate symbol. This same procedure can be followed for each pair of symbols in Figure 3-33.

Several points should be stressed regarding the logic symbol equivalences:

1. The equivalences can be extended to gates with *any* number of inputs.
2. None of the standard symbols have bubbles on their inputs, and all the alternate symbols do.

3. The standard and alternate symbols for each gate represent the same physical circuit; *there is no difference in the circuits represented by the two symbols.*
4. NAND and NOR gates are inverting gates, and so both the standard and the alternate symbols for each will have a bubble on *either* the input or the output. AND and OR gates are *noninverting* gates, and so the alternate symbols for each will have bubbles on *both* inputs and output.

Logic-Symbol Interpretation

Each of the logic-gate symbols of Figure 3-33 provides a unique interpretation of how the gate operates. Before we can demonstrate these interpretations, we must first establish the concept of **active logic levels.**

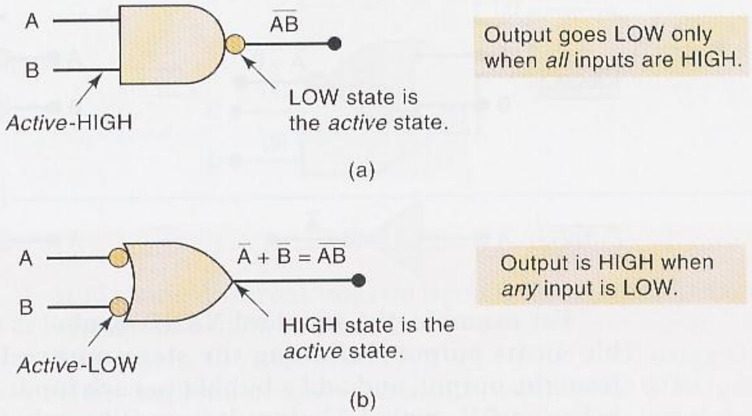
When an input or output line on a logic circuit symbol has *no bubble* on it, that line is said to be **active-HIGH**. When an input or output line *does* have a *bubble* on it, that line is said to be **active-LOW**. The presence or absence of a bubble, then, determines the active-HIGH/active-LOW status of a circuit's inputs and output, and is used to interpret the circuit operation.

To illustrate, Figure 3-34(a) shows the standard symbol for a NAND gate. The standard symbol has a bubble on its output and no bubbles on its inputs. Thus, it has an active-LOW output and active-HIGH inputs. The logic operation represented by this symbol can therefore be interpreted as follows:

The output goes LOW only when *all* of the inputs are HIGH.

Note that this says that the output will go to its active state only when *all* of the inputs are in their active states. The word *all* is used because of the AND symbol.

FIGURE 3-34
Interpretation of the two NAND gate symbols.


A
B
$\overline{AB}$
Active-HIGH
LOW state is the *active* state.
(a)
A
B
$\overline{A}+\overline{B}=\overline{AB}$
Active-LOW
HIGH state is the *active* state.
(b)

Output goes LOW only when *all* inputs are HIGH.

Output is HIGH when *any* input is LOW.

The alternate symbol for a NAND gate shown in Figure 3-34(b) has an active-HIGH output and active-LOW inputs, and so its operation can be stated as follows:

The output goes HIGH when *any input* is LOW.

This says that the output will be in its active state whenever *any* of the inputs is in its active state. The word *any* is used because of the OR symbol.

With a little thought, you can see that the two interpretations for the NAND symbols in Figure 3-34 are different ways of saying the same thing.

Summary

At this point you are probably wondering why there is a need to have two different symbols and interpretations for each logic gate. We hope the reasons will become clear after reading the next section. For now, let us summarize the important points concerning the logic-gate representations.

1. To obtain the alternate symbol for a logic gate, take the standard symbol and change its operation symbol (OR to AND, or AND to OR), and change the bubbles on both inputs and output (i.e., delete bubbles that are present, and add bubbles where there are none).
2. To interpret the logic-gate operation, first note which logic state, 0 or 1, is the active state for the inputs and which is the active state for the output. Then realize that the output's active state is produced by having *all* of the inputs in their active state (if an AND symbol is used) or by having *any* of the inputs in its active state (if an OR symbol is used).

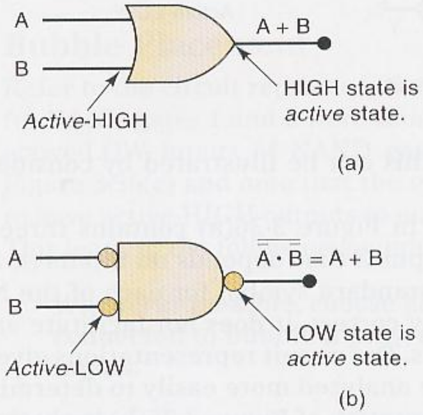
EXAMPLE 3-19

Give the interpretation of the two OR gate symbols.

Solution

The results are shown in Figure 3-35. Note that the word *any* is used when the operation symbol is an OR symbol and the word *all* is used when it includes an AND symbol.

FIGURE 3-35 Interpretation of the two OR gate symbols.


A
B
A + B
Active-HIGH
HIGH state is active state.
(a)
A
B
$\overline{A \cdot B} = A + B$
Active-LOW
LOW state is active state.
(b)

Output goes HIGH when *any* input is HIGH.

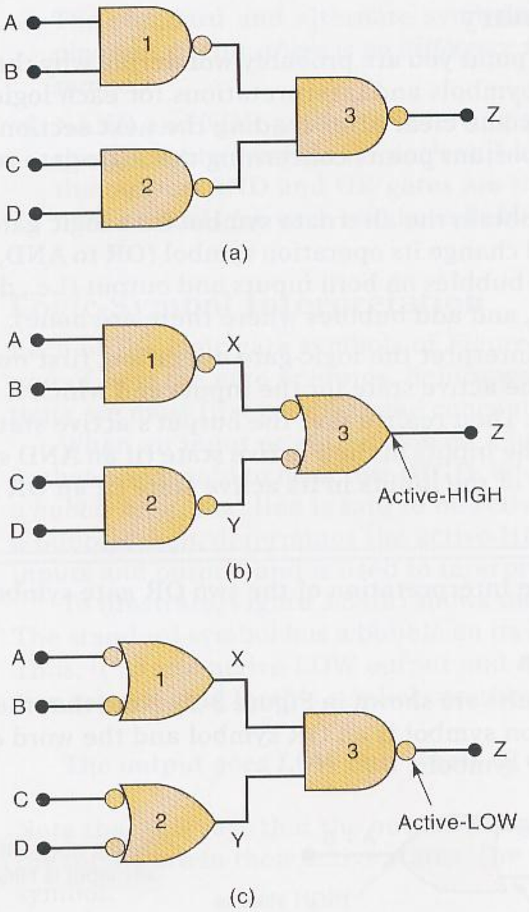
Output goes LOW only when *all* inputs are LOW.

REVIEW QUESTIONS

1. Write the interpretation of the operation performed by the standard NOR gate symbol in Figure 3-33.
2. Repeat question 1 for the alternate NOR gate symbol.
3. Repeat question 1 for the alternate AND gate symbol.
4. Repeat question 1 for the standard AND gate symbol.

3-14 WHICH GATE REPRESENTATION TO USE

Some logic-circuit designers and some textbooks use only the standard logic-gate symbols in their circuit schematics. While this practice is not incorrect, it does nothing to make the circuit operation easier to follow. Proper use of the alternate gate symbols in the circuit diagram can make the circuit operation



A	B	C	D	Z
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

(d)

FIGURE 3-36 (a) Original circuit using standard NAND symbols; (b) equivalent representation where output Z is active-HIGH; (c) equivalent representation where output Z is active-LOW; (d) truth table.

much clearer. This can be illustrated by considering the example shown in Figure 3-36.

The circuit in Figure 3-36(a) contains three NAND gates connected to produce an output Z that depends on inputs A, B, C, and D. The circuit diagram uses the standard symbol for each of the NAND gates. While this diagram is logically correct, it does not facilitate an understanding of how the circuit functions. The circuit representations given in Figures 3-36(b) and (c), however, can be analyzed more easily to determine the circuit operation.

The representation of Figure 3-36(b) is obtained from the original circuit diagram by replacing NAND gate 3 with its alternate symbol. In this diagram, output Z is taken from a NAND gate symbol that has an active-HIGH output. Thus, we can say that Z will go HIGH when either X or Y is LOW. Now, since X and Y each appear at the output of NAND symbols having active-LOW outputs, we can say that X will go LOW only if $A = B = 1$, and Y will go LOW only if $C = D = 1$. Putting this all together, we can describe the circuit operation as follows:

Output Z will go HIGH whenever either $A = B = 1$ or $C = D = 1$ (or both).

This description can be translated to truth-table form by setting $Z = 1$ for those cases where $A = B = 1$ and for those cases where $C = D = 1$. For all other cases, Z is made a 0. The resultant truth table is shown in Figure 3-36(d).

The representation of Figure 3-36(c) is obtained from the original circuit diagram by replacing NAND gates 1 and 2 by their alternate symbols. In this equivalent representation, the Z output is taken from a NAND gate that has an active-LOW output. Thus, we can say that Z will go LOW only when $X = Y = 1$. Because X and Y are active-HIGH outputs, we can say that X will be HIGH when either A or B is LOW, and Y will be HIGH when either C or D is LOW. Putting this all together, we can describe the circuit operation as follows:

Output Z will go LOW only when A or B is LOW and C or D is LOW.

This description can be translated to truth-table form by making $Z = 0$ for all cases where at least one of the A or B inputs is LOW at the same time that at least one of the C or D inputs is LOW. For all other cases, Z is made a 1. The resultant truth table is the same as that obtained for the circuit diagram of Figure 3-36(b).

Which Circuit Diagram Should Be Used?

The answer to this question depends on the particular function being performed by the circuit output. If the circuit is being used to cause some action (e.g., turn on an LED or activate another logic circuit) when output Z goes to the 1 state, then we say that Z is to be active-HIGH, and the circuit diagram of Figure 3-36(b) should be used. On the other hand, if the circuit is being used to cause some action when Z goes to the 0 state, then Z is to be active-LOW, and the diagram of Figure 3-36(c) should be used.

Of course, there will be situations where *both* output states are used to produce different actions and either one can be considered to be the active state. For these cases, either circuit representation can be used.

Bubble Placement

Refer to the circuit representation of Figure 3-36(b) and note that the symbols for NAND gates 1 and 2 were chosen to have active-LOW outputs to match the active-LOW inputs of NAND gate 3. Refer to the circuit representation of Figure 3-36(c) and note that the symbols for NAND gates 1 and 2 were chosen to have active-HIGH outputs to match the active-HIGH inputs of NAND gate 3. This leads to the following general rule for preparing logic-circuit schematics:

Whenever possible, choose gate symbols so that bubble outputs are connected to bubble inputs, and nonbubble outputs to nonbubble inputs.

The following examples will show how this rule can be applied.

EXAMPLE 3-20

The logic circuit in Figure 3-37(a) is being used to activate an alarm when its output Z goes HIGH. Modify the circuit diagram so that it represents the circuit operation more effectively.

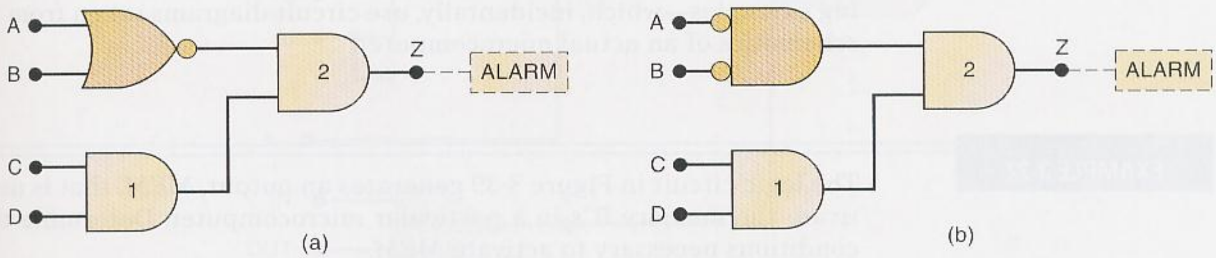


FIGURE 3-37 Example 3-20.

Solution

Because Z = 1 will activate the alarm, Z is to be active-HIGH. Thus, the AND gate 2 symbol does not have to be changed. The NOR gate symbol should be changed to the alternate symbol with a nonbubble (active-HIGH) output to match the nonbubble input of AND gate 2, as shown in Figure 3-37(b). Note that the circuit now has nonbubble outputs connected to the nonbubble inputs of gate 2.

EXAMPLE 3-21

When the output of the logic circuit in Figure 3-38(a) goes LOW, it activates another logic circuit. Modify the circuit diagram to represent the circuit operation more effectively.

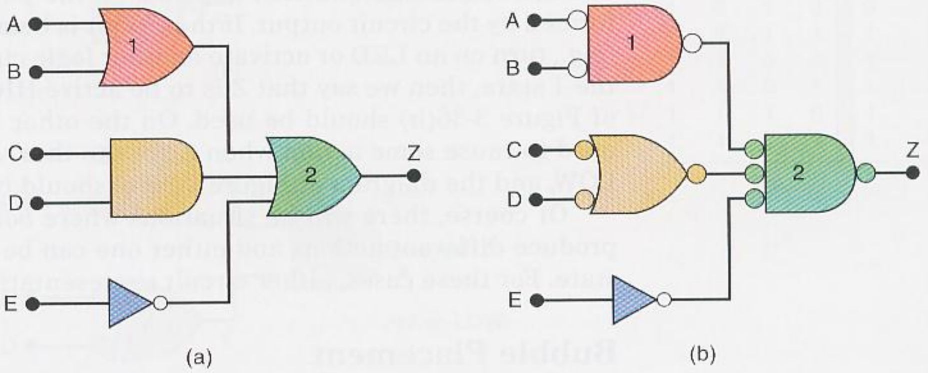


FIGURE 3-38 Example 3-21.

Solution

Because Z is to be active-LOW, the symbol for OR gate 2 must be changed to its alternate symbol, as shown in Figure 3-38(b). The new OR gate 2 symbol has bubble inputs, and so the AND gate and OR gate 1 symbols must be changed to bubbled outputs, as shown in Figure 3-38(b). The INVERTER already has a bubble output. Now the circuit has all bubble outputs connected to bubble inputs of gate 2.

Analyzing Circuits

When a logic-circuit schematic is drawn using the rules we followed in these examples, it is much easier for an engineer or technician (or student) to follow the signal flow through the circuit and to determine the input conditions that are needed to activate the output. This will be illustrated in the following examples—which, incidentally, use circuit diagrams taken from the logic schematics of an actual microcomputer.

EXAMPLE 3-22

The logic circuit in Figure 3-39 generates an output, *MEM*, that is used to activate the memory ICs in a particular microcomputer. Determine the input conditions necessary to activate *MEM*.

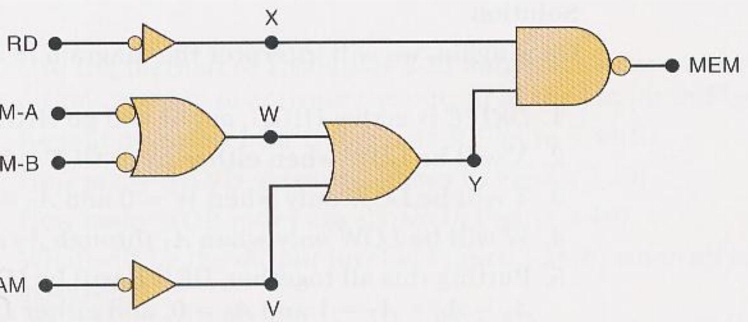


FIGURE 3-39
Example 3-22.

Solution

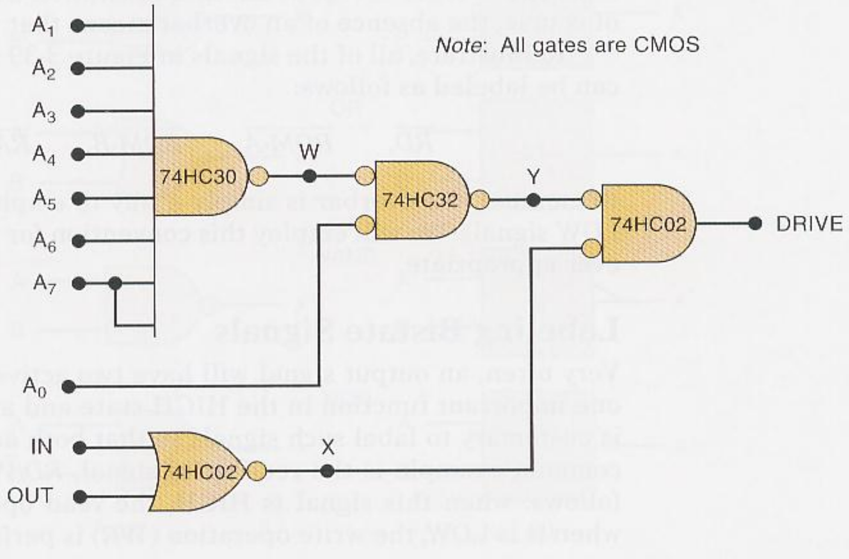
One way to do this would be to write the expression for *MEM* in terms of the inputs *RD*, *ROM-A*, *ROM-B*, and *RAM*, and to evaluate it for the 16 possible combinations of these inputs. While this method would work, it would require a lot more work than is necessary.

A more efficient method is to interpret the circuit diagram using the ideas we have been developing in the last two sections. These are the steps:

1. *MEM* is active-LOW, and it will go LOW only when X and Y are HIGH.
2. X will be HIGH only when $RD = 0$.
3. Y will be HIGH when either W or V is HIGH.
4. V will be HIGH when $RAM = 0$.
5. W will be HIGH when either *ROM-A* or $ROM\text{-}B = 0$.
6. Putting this all together, *MEM* will go LOW only when $RD = 0$ *and* at least one of the three inputs *ROM-A*, *ROM-B*, or *RAM* is LOW.

EXAMPLE 3-23

The logic circuit in Figure 3-40 is used to control the drive spindle motor for a floppy disk drive when the microcomputer is sending data to or receiving data from the disk. The circuit will turn on the motor when $DRIVE = 1$. Determine the input conditions necessary to turn on the motor.



Note: All gates are CMOS

FIGURE 3-40
Example 3-23.

Solution

Once again, we will interpret the diagram in a step-by-step fashion:

1. *DRIVE* is active-HIGH, and it will go HIGH only when $X = Y = 0$.
2. X will be LOW when either *IN* or *OUT* is HIGH.
3. Y will be LOW only when $W = 0$ and $A_0 = 0$.
4. W will be LOW only when A_1 through A_7 are all HIGH.
5. Putting this all together, *DRIVE* will be HIGH when $A_1 = A_2 = A_3 = A_4 = A_5 = A_6 = A_7 = 1$ and $A_0 = 0$, and either *IN* or *OUT* or both are 1.

Note the strange symbol for the eight-input CMOS NAND gate (74HC30); also note that signal A_7 is connected to two of the NAND inputs.

Asserted Levels

We have been describing logic signals as being active-LOW or active-HIGH. For example, the output *MEM* in Figure 3-39 is active-LOW, and the output *DRIVE* in Figure 3-40 is active-HIGH because these are the output states that cause something to happen. Similarly, Figure 3-40 has active-HIGH inputs A_1 to A_7, and active-LOW input A_0.

When a logic signal is in its active state, it can be said to be **asserted**. For example, when we say that input A_0 is asserted, we are saying that it is in its active-LOW state. When a logic signal is not in its active state, it is said to be **unasserted**. Thus, when we say that *DRIVE* is unasserted, we mean that it is in its inactive state (low).

Clearly, the terms *asserted* and *unasserted* are synonymous with *active* and *inactive*, respectively:

$$\text{asserted} = \text{active}$$
$$\text{unasserted} = \text{inactive}$$

Both sets of terms are in common use in the digital field, so you should recognize both ways of describing a logic signal's active state.

Labeling Active-LOW Logic Signals

It has become common practice to use an overbar to label active-LOW signals. The overbar serves as another indication that the signal is active-LOW; of course, the absence of an overbar means that the signal is active-HIGH.

To illustrate, all of the signals in Figure 3-39 are active-LOW, and so they can be labeled as follows:

$$\overline{RD}, \quad \overline{ROM\text{-}A}, \quad \overline{ROM\text{-}B}, \quad \overline{RAM}, \quad \overline{MEM}$$

Remember, the overbar is simply a way to emphasize that these are active-LOW signals. We will employ this convention for labeling logic signals whenever appropriate.

Labeling Bistate Signals

Very often, an output signal will have two active states; that is, it will have one important function in the HIGH state and another in the LOW state. It is customary to label such signals so that both active states are apparent. A common example is the read/write signal, $RD/\overline{WR}$, which is interpreted as follows: when this signal is HIGH, the read operation (*RD*) is performed; when it is LOW, the write operation (*WR*) is performed.

REVIEW QUESTIONS

1. Use the method of Examples 3-22 and 3-23 to determine the input conditions needed to activate the output of the circuit in Figure 3-37(b).
2. Repeat question 1 for the circuit of Figure 3-38(b).
3. How many NAND gates are shown in Figure 3-39?
4. How many NOR gates are shown in Figure 3-40?
5. What will be the output level in Figure 3-38(b) when all of the inputs are asserted?
6. What inputs are required to assert the alarm output in Figure 3-37(b)?
7. Which of the following signals is active-LOW: RD, $\overline{W}$, $R/\overline{W}$?

3-15 IEEE/ANSI STANDARD LOGIC SYMBOLS

The logic symbols we have used so far in this chapter are the *traditional* standard symbols used in the digital industry for many, many years. These traditional symbols use a distinctive shape for each logic gate. A newer standard for logic symbols was developed in 1984; it is called the **IEEE/ANSI** Standard 91-1984 for logic symbols. The IEEE/ANSI standard uses rectangular symbols to represent all logic gates and circuits. A special *dependency notation* inside the rectangular symbol indicates how the device outputs depend on the device inputs. Figure 3-41 shows the IEEE/ANSI symbols alongside the traditional symbols for the basic logic gates. Note the following points:

1. The rectangular symbols use a small right triangle (◺) in place of the small bubble of the traditional symbols to indicate the inversion of the logic level. The presence or absence of the triangle also signifies whether an input or output is active-LOW or active-HIGH.

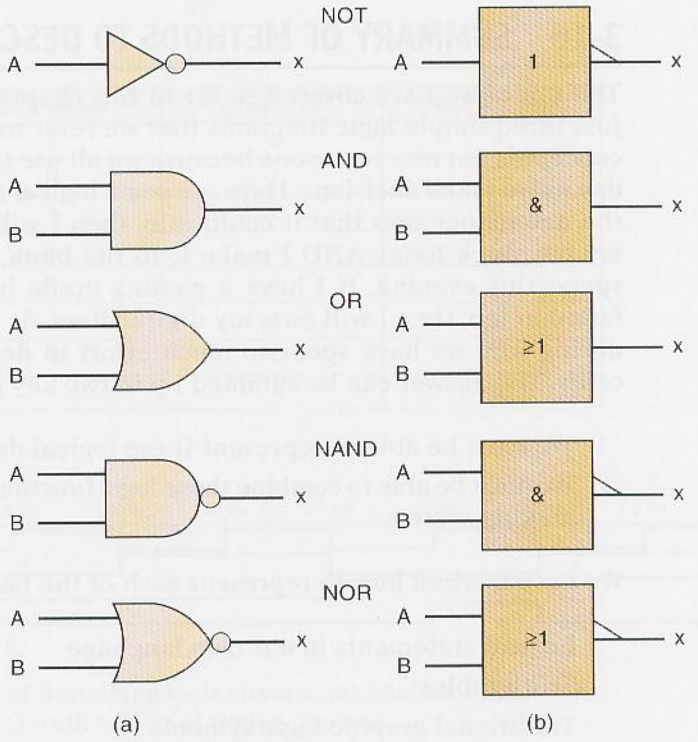


FIGURE 3-41 Standard logic symbols: (a) traditional; (b) IEEE/ANSI.

2. A special notation inside each rectangular symbol describes the logic relation between inputs and output. The "1" inside the INVERTER symbol denotes a device with only *one* input; the triangle on the output indicates that the output will go to its active-LOW state when that one input is in its active-HIGH state. The "&" inside the AND symbol means that the output will go to its active-HIGH state when all of the inputs are in their active-HIGH state. The "≥" inside the OR gate means that the output will go to its active state (HIGH) whenever *one or more* inputs are in their active state (HIGH).
3. The rectangular symbols for the NAND and the NOR are the same as those for the AND and the OR, respectively, with the addition of the small inversion triangle on the output.

Traditional or IEEE/ANSI?

The IEEE/ANSI standard has not yet been widely accepted for use in the digital field, although you will run across it in some newer equipment schematics. Most digital IC data books include both the traditional and IEEE/ANSI symbols, and it is possible that the newer standard might eventually become more widely used. We will employ the traditional symbols in most of the circuit diagrams throughout this book.

REVIEW QUESTIONS

1. Draw all of the basic logic gates using both the traditional symbols and the IEEE/ANSI symbols.
2. Draw the IEEE/ANSI symbol for a NOR gate with active-HIGH output.

3-16 SUMMARY OF METHODS TO DESCRIBE LOGIC CIRCUITS

The topics we have covered so far in this chapter have all centered around just three simple logic functions that we refer to as AND, OR, and NOT. The concept is not new to anyone because we all use these logical functions every day as we make decisions. Here are some logical examples. If it is raining OR the newspaper says that it could rain, then I will take my umbrella. If I get my paycheck today AND I make it to the bank, then I will have money to spend this evening. If I have a passing grade in lecture AND I have NOT failed in lab, then I will pass my digital class. At this point, you may be wondering why we have spent so much effort in describing such familiar concepts. The answer can be summed up in two key points:

1. We must be able to represent these logical decisions.
2. We must be able to combine these logic functions and implement a decision-making system.

We have learned how to represent each of the basic logic functions using:

Logical statements in our own language
Truth tables
Traditional graphic logic symbols

IEEE/ANSI standard logic symbols
Boolean algebra expressions
Timing diagrams

EXAMPLE 3-24

The following English expression describes the way a logic circuit needs to operate in order to drive a seatbelt warning indicator in a car.

> **If the driver is present AND the driver is NOT buckled up AND the ignition switch is on, THEN turn on the warning light.**

Describe the circuit using Boolean algebra, schematic diagrams with logic symbols, truth tables, and timing diagrams.

Solution

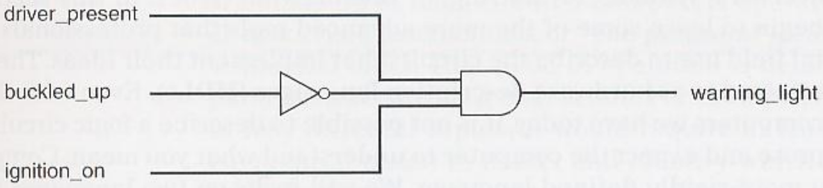
See Figure 3-42.

Boolean expression

$\text{warning\_light} = \text{driver\_present} \bullet \overline{\text{buckled\_up}} \bullet \text{ignition\_on}$

(a)

Schematic diagram



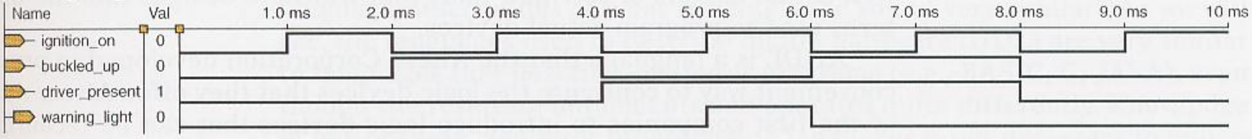
(b)

Truth table

driver_present	buckled_up	ignition_on	warning_light
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

(c)

Timing diagram



(d)

FIGURE 3-42 Methods of describing logic circuits: (a) Boolean expression; (b) schematic diagram; (c) truth table; (d) timing diagram.

Figure 3-42 shows four different ways of representing the logic circuit that was described in English as the problem statement of Example 3-24. There are many other ways in which we could represent the logic of this decision. As an example we could dream up an entirely new set of graphic symbols, or state the logical relationship in French or Japanese. Of course, we cannot cover all the possible ways of describing a logic circuit, but we must understand the most common methods to be able to communicate with others in this profession. Furthermore, certain situations are easier to describe using one method over another. In some cases, a picture is worth a thousand words, and in other cases words are concise enough and are more easily communicated to others. The important point here is that we need ways to describe and communicate the operation of digital systems.

REVIEW QUESTION

1. Name five ways to describe the operation of logic circuits.

3-17 DESCRIPTION LANGUAGES VERSUS PROGRAMMING LANGUAGES*

Recent trends in the field of digital systems are favoring text-based language description of digital circuits. You probably noticed that each description method in Figure 3-42 offers challenges to computer entry, whether it is due to overbars, symbols, formatting, or line-drawing issues. In this section, we will begin to learn some of the more advanced tools that professionals in the digital field use to describe the circuits that implement their ideas. These tools are referred to as **hardware description languages (HDLs)**. Even with the powerful computers we have today, it is not possible to describe a logic circuit in English prose and expect the computer to understand what you mean. Computers need a more rigidly defined language. We will focus on two languages in this text: **Altera hardware description language (AHDL)** and **very high speed integrated circuit (VHSIC) hardware description language (VHDL)**.

VHDL and AHDL

VHDL is not a new language. It was developed by the Department of Defense in the early 1980s as a concise way to document the designs in the very high speed integrated circuit (VHSIC) program. Appending HDL onto this acronym was too much, even for the military, and so the language was abbreviated to VHDL. Computer programs were developed to take the VHDL language files and simulate the operation of the circuits. With the growth of complex programmable logic devices in digital systems, VHDL has evolved into one of the primary high-level hardware description languages for designing and implementing digital circuits (synthesis). The language has been standardized by the IEEE, making it universally appealing for engineers as well as the makers of software tools that translate designs into the bit patterns used to program actual devices.

AHDL is a language that the Altera Corporation developed to provide a convenient way to configure the logic devices that they offer. Altera was one of the first companies to introduce logic devices that can be reconfigured electronically. These devices are called **programmable logic devices (PLDs)**. Unlike VHDL, this language is not intended to be used as a universal language for describing any logic circuit. It is intended to be used for programming complex digital systems into Altera PLDs in a language that is generally perceived to be easier to learn yet very similar to VHDL. It also has features that take full advantage of the architecture of Altera devices. All of the examples in this text will use the Altera MAX+PLUS II or Quartus II software to develop both AHDL and VHDL design files. You will see the advantage of using Altera's development system for both languages when you program an actual device. The Altera system makes circuit development very easy and contains all the necessary tools to translate from the HDL design file to a file ready to load into an Altera PLD. It also allows you to develop building blocks using schematic entry, AHDL, VHDL, and other methods and then interconnect them to form a complete system.

---

*All sections covering hardware description languages may by skipped without loss of continuity in the balance of Chapters 1–12.

Other HDLs are available that are more suitable for programming simple programmable logic devices. You will find any of these languages easy to use after learning the basics of AHDL or VHDL as covered in this text.

Computer Programming Languages

It is important to distinguish between hardware description languages intended to describe the hardware configuration of a circuit and programming languages that represent a sequence of instructions intended to be carried out by a computer to accomplish some task. In both cases, we use a *language* to *program* a device. However, computers are complex digital systems that are made up of logic circuits. Computers operate by following a laundry list of tasks (i.e., instructions, or "the program"), each of which must be done in sequential order. The speed of operation is determined by how fast the computer can execute each instruction. For example, if a computer were to respond to four different inputs, it would require at least four separate instructions (sequential tasks) to detect and identify which input changed state. A digital logic circuit, on the other hand, is limited in its speed only by how quickly the circuitry can change the outputs in response to changes in the inputs. It is monitoring all inputs **concurrently** (at the same time) and responding to any changes.

The following analogy will help you understand the difference between computer operation and digital logic circuit operation and the role of language elements used to describe what the systems do. Consider the challenge of describing what is done to an Indy 500 car during a pit stop. If a single person performed all the necessary tasks one at a time, he or she would need to be very fast at each task. This is the way a computer operates: one task at a time but very quickly. Of course, at Indy, there is an entire pit crew that swarms the car, and each member of the crew does his or her task while the others do theirs. All crew members operate concurrently, like the elements of a digital circuit. Now consider how you would describe to someone else what is being done to the Indy car during the pit stop using (1) the individual-mechanic approach or (2) the pit-crew approach. Wouldn't the two English language descriptions of what is being done sound very similar? As we will see, the languages used to describe digital hardware (HDL) are very similar to languages that describe computer programs (e.g., BASIC, C, JAVA), even though the resulting implementation operates quite differently. Knowledge of any of these computer programming languages is not necessary to understand HDL. The important thing is that when you have learned both an HDL and a computer language, you must understand their different roles in digital systems.

EXAMPLE 3-25

Compare the operation of a computer and a logic circuit in performing the simple logical operation of $y = AB$.

Solution

The logic circuit is a simple AND gate. The output y will be HIGH within approximately 10 nanoseconds of the point when A and B are HIGH simultaneously. Within approximately 10 nanoseconds after either input goes LOW, the output y will be LOW.

The computer must run a program of instructions that makes decisions. Suppose each instruction takes 20 ns (that's pretty fast!). Each shape in the flowchart shown in Figure 3-43 represents one instruction. Clearly, it will take a minimum of two or three instructions (40–60 ns) to respond to changes in the inputs.

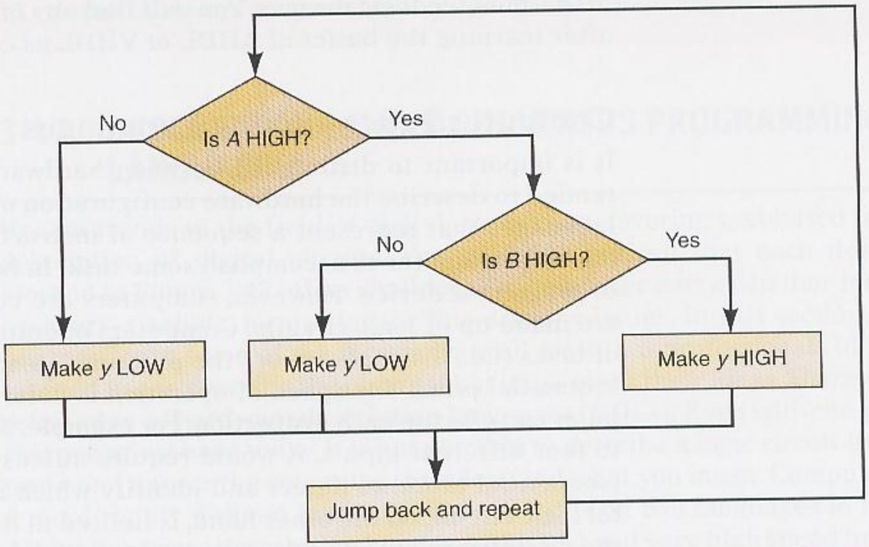


FIGURE 3-43 Decision process of a computer program.

REVIEW QUESTIONS

1. What does HDL stand for?
2. What is the purpose of an HDL?
3. What is the purpose of a computer programming language?
4. What is the key difference between HDL and computer programming languages?

3-18 IMPLEMENTING LOGIC CIRCUITS WITH PLDs

Many digital circuits today are implemented using programmable logic devices (PLDs). These devices are not like microcomputers or microcontrollers that "run" the program of instructions. Instead, they are configured electronically, and their internal circuits are "wired" together electronically to form a logic circuit. This programmable wiring can be thought of as thousands of connections that are either connected (1) or not connected (0). Figure 3-44 shows a small area of programmable connections. Each intersection between a row (horizontal wire) and a column (vertical wire) is a programmable connection. You can imagine how difficult it would be to try to

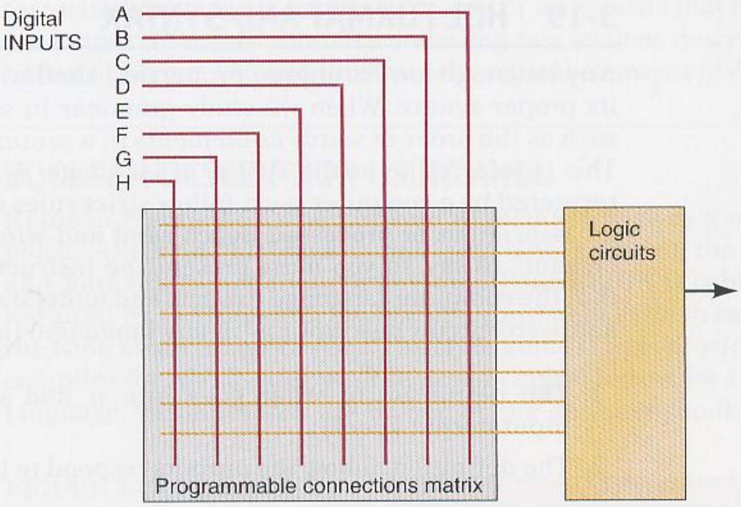


FIGURE 3-44 Configuring hardware connections with programmable logic devices.

configure these devices by placing 1s and 0s in a grid manually (which is how they did it back in the 1970s).

The role of the hardware description language is to provide a concise and convenient way for the designer to describe the operation of the circuit in a format that a personal computer can handle and store conveniently. The computer runs a special software application called a **compiler** to translate from the hardware description language into the grid of 1s and 0s that can be loaded into the PLD. If a person can master the higher-level hardware description language, it actually makes programming the PLDs much easier than trying to use Boolean algebra, schematic drawings, or truth tables. In much the same way that you learned the English language, we will start by expressing simple things and gradually learn the more complicated aspects of these languages. Our objective is to learn enough of HDL to be able to communicate with others and perform simple tasks. A full understanding of all the details of these languages is beyond the scope of this text and can really be mastered only by regular use.

In the sections throughout this book that cover the HDLs, we will present both AHDL and VHDL in a format that allows you to skip over one language and concentrate on the other without missing important information. Of course, this setup means there will be some redundant information presented if you choose to read about both languages. We feel this redundancy is worth the extra effort to provide you with the flexibility of focusing on either of the two languages or learning both by comparing and contrasting similar examples. The recommended way to use the text is to focus on one language. It is true that the easiest way to become bilingual, and fluent in both languages, is to be raised in an environment where both languages are spoken routinely. It is also very easy, however, to confuse details, so we will keep the specific examples separate and independent. We hope this format provides you with the opportunity to learn one language now and then use this book as a reference later in your career should you need to pick up the second language.

REVIEW QUESTIONS

1. What does PLD stand for?
2. How are the circuits reconfigured electronically in a PLD?
3. What does a compiler do?

3-19 HDL FORMAT AND SYNTAX

Any language has its unique properties, similarities to other languages, and its proper syntax. When we study grammar in school, we learn conventions such as the order of words as elements in a sentence and proper punctuation. This is referred to as the **syntax** of language. A language designed to be interpreted by a computer must follow strict rules of syntax. A computer is just an assortment of processed beach sand and wire that has no idea what you "meant" to say, so you must present the instructions using the exact syntax that the computer language expects and understands. The basic format of any hardware circuit description (in any language) involves two vital elements:

1. The definition of what goes into it and what comes out of it (i.e., input/output specs)
2. The definition of how the outputs respond to the inputs (i.e., its operation)

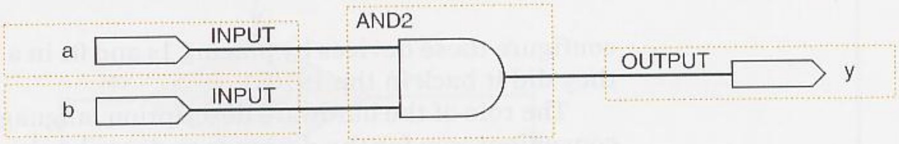


FIGURE 3-45 A schematic diagram description.

A circuit schematic diagram such as Figure 3-45 can be read and understood by a competent engineer or technician because both would understand the meaning of each symbol in the drawing. If you understand how each element works and how the elements are connected to each other, you can understand how the circuit operates. On the left side of the diagram is the set of inputs, and on the right is the set of outputs. The symbols in the middle define its operation. The text-based language must convey the same information. All HDLs use the format shown in Figure 3-46.

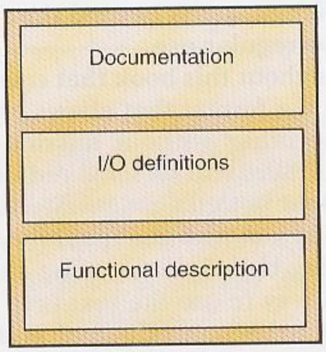


FIGURE 3-46 Format of HDL files.

In a text-based language, the circuit being described must be given a name. The inputs and outputs (sometimes called ports) must be assigned names and defined according to the nature of the port. Is it a single bit from a toggle switch? Or is it a four-bit number coming from a keypad? The text-based language must somehow convey the nature of these inputs and outputs. The **mode** of a port defines whether it is input, output, or both. The **type** refers to the number of bits and how those bits are grouped and interpreted. If the *type* of input is a single bit, then it can have only two possible values: 0 and 1. If the type of input is a four-bit binary number from a keypad, it can have any one of 16 different values (0000_2–1111_2). The type determines the range of possible values. The definition of the circuit's operation in a text-based language is contained in a set of statements that follow the circuit input/output (I/O) definition. The following two sections describe the very simple circuit of Figure 3-45 and illustrate the critical elements of AHDL and VHDL.

BOOLEAN DESCRIPTION USING AHDL

Refer to Figure 3-47. The keyword **SUBDESIGN** gives a name to the circuit block, which in this case is *and_gate*. The name of the file must also be and_gate.tdf. Notice that the keyword SUBDESIGN is capitalized. This is not required by the software, but use of a consistent style in capitalization makes the code much easier to read. The style guide that is provided with the Altera compiler for AHDL suggests the use of capital letters for the keywords in the language. Variables that are named by the designer should be lowercase.

FIGURE 3-47 Essential elements in AHDL.

```
SUBDESIGN and_gate
(
   a, b        :INPUT;
   y           :OUTPUT;
)
BEGIN
   y = a & b;
END;
```

The SUBDESIGN section defines the inputs and outputs of the logic circuit block. Something must enclose the circuit that we are trying to describe, much the same way that a block diagram encloses everything that makes up that part of the design. In AHDL, this input/output definition is enclosed in parentheses. The list of variables used for inputs to this block are separated by commas and followed by :INPUT;. In AHDL, the single-bit type is assumed unless the variable is designated as multiple bits. The single-output bit is declared with the mode :OUTPUT;. We will learn the proper way to describe other types of inputs, outputs, and variables as we need to use them.

The set of statements that describe the operation of the AHDL circuit are contained in the logic section between the keywords BEGIN and END. In this example, the operation of the hardware is described by a very simple Boolean algebra equation that states that the output (y) is assigned (=) the logic level produced by a AND b. This Boolean algebra equation is referred to as a **concurrent assignment statement**. Any statements (there is only one in this example) between BEGIN and END are evaluated constantly and concurrently. The order in which they are listed makes no difference. The basic Boolean operators are:

&	AND
#	OR
!	NOT
$	XOR

REVIEW QUESTIONS

1. What appears inside the parentheses ( ) after SUBDESIGN?
2. What appears between BEGIN and END?

BOOLEAN DESCRIPTION USING VHDL

Refer to Figure 3-48. The keyword **ENTITY** gives a name to the circuit block, which in this case is and_gate. Notice that the keyword ENTITY is capitalized but and_gate is not. This is not required by the software, but use of a consistent style in capitalization makes the code much easier to read. The style guide provided with the Altera compiler for VHDL suggests using capital letters for the keywords in the language. Variables that are named by the designer should be lowercase.

FIGURE 3-48 Essential elements in VHDL.

```
ENTITY and_gate IS
PORT (     a, b     :IN BIT;
               y      :OUT BIT);
END and_gate;
ARCHITECTURE ckt OF and_gate IS
BEGIN
          y <= a AND b;
END ckt;
```

The ENTITY declaration can be thought of as a block description. Something must enclose the circuit we are trying to describe, much the same way a block diagram encloses everything that makes up that part of the design. In VHDL, the keyword PORT tells the compiler that we are defining inputs and outputs to this circuit block. The names used for inputs (separated by commas) are listed, ending with a colon and a description of the mode and type of input (:IN BIT;). In VHDL, the **BIT** description tells the compiler that each variable in the list is a single bit. We will learn the proper way to describe other types of inputs, outputs, and variables as we need to use them. The line containing END and_gate; terminates the ENTITY declaration.

The **ARCHITECTURE** declaration is used to describe the operation of everything inside the block. The designer makes up a name for this architectural description of the inner workings of the ENTITY block (*ckt* in this example). Every ENTITY must have at least one ARCHITECTURE associated with it. The words OF and IS are keywords in this declaration. The body of the architecture description is enclosed between the BEGIN and END keywords. END is followed by the name that has been assigned to this architecture. Within the body (between BEGIN and END) is the description of the block's operation. In this example, the operation of the hardware is described by a very simple Boolean algebra equation that states that the output (y) is assigned ($<=$) the logic level produced by a AND b. This is referred to as a **concurrent assignment statement**, which means that all the statements (there is only one in this example) between BEGIN and END are evaluated constantly and concurrently. The order in which they are listed makes no difference.

REVIEW QUESTIONS

1. What is the role of the ENTITY declaration?
2. Which key section defines the operation of the circuit?
3. What is the assignment operator used to give a value to a logic signal?

3-20 INTERMEDIATE SIGNALS

In many designs, there is a need to define signal points "inside" the circuit block. They are points in the circuit that are neither inputs nor outputs for the block but may be useful as a reference point. It may be a signal that needs to be connected to many other places within the block. In an analog or digital schematic diagram, they would be called test points or *nodes*. In an HDL, they are referred to as **buried nodes** or **local signals**. Figure 3-49 shows a very simple circuit that uses an intermediate signal named *m*. In the HDL, these nodes (signals) are not defined with the inputs and outputs but rather in the section that describes the operation of the block. The inputs and outputs are available to other circuit blocks in the system, but these local signals are recognized only within this block.

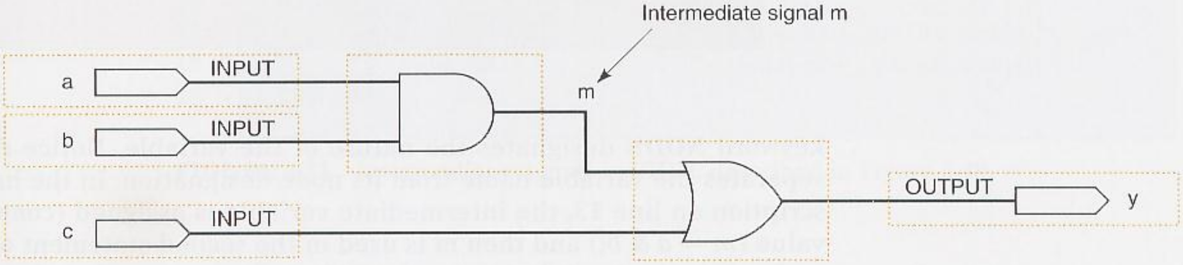


FIGURE 3-49 A logic circuit diagram with an intermediate variable.

In the example code that follows, notice the information at the top. The purpose of this information is strictly for documentation purposes. It is absolutely vital that the design is documented thoroughly. At a minimum, it should describe the project it is being used in, who wrote it, and the date. This information is often referred to as a header. We are keeping our headers brief to make this book a little lighter to carry to class, but remember: memory space is cheap and information is valuable. So don't be afraid to *document thoroughly!* There are also comments next to many of the statements in the code. These comments help the designer remember what she or he was trying to do and to help any other person to understand what was intended.

AHDL BURIED NODES

The AHDL code that describes the circuit in Figure 3-49 is shown in Figure 3-50. The **comments** in AHDL can be enclosed between % characters, as you can see in the figure between lines 1 and 4. This section of the code allows the designer to write many lines of information that will be ignored by computer programs using this file but can be read by any person trying to decipher the code. Notice that the comments at the end of lines 9, 10, 13, 15, and 16 are preceded by two dashes (--). The text following the dashes is for documentation only. Either type of comment symbol may be used, but percent signs must be used in pairs to open and close a comment. Double dashes indicate a comment that extends to the end of the line.

In AHDL, local signals are declared in the VARIABLE section, which is placed between the SUBDESIGN section and the logic section. The intermediate signal *m* is defined on line 11, following the keyword **VARIABLE**. The

FIGURE 3-50 Intermediate variables in AHDL described in Figure 3-49.

```
 1  %  Intermediate variables in AHDL (Figure 3-49)
 2     Digital Systems 10th ed
 3     NS Widmer
 4     MAY 23, 2005          %
 5  SUBDESIGN fig3_50
 6  (
 7     a,b,c      :INPUT;    -- define inputs to block
 8     y          :OUTPUT;   -- define block output
 9  )
10  VARIABLE
11     m        :NODE;       -- name an intermediate signal
12  BEGIN
13     m = a & b;            -- generate buried product term
14     y = m # c;            -- generate sum on output
15  END;
```

keyword **NODE** designates the nature of the variable. Notice that a colon separates the variable name from its node designation. In the hardware description on line 13, the intermediate variable is assigned (connected to) a value ($m = a$ & b;) and then m is used in the second statement on line 14 to assign (connect) a value to y ($y = m$ # c;). Remember that the assignment statements are concurrent and, thus, the order in which they are given does not matter. For human readability, it may seem more logical to assign values to intermediate variables before they are used in other assignment statements, as shown here.

REVIEW QUESTIONS

1. What is the designation used for intermediate variables?
2. Where are these variables declared?
3. Does it matter whether the m or y equation comes first?
4. What character is used to limit a block of comments?
5. What characters are used to comment a single line?

VHDL LOCAL SIGNALS

The VHDL code that describes the circuit in Figure 3-49 is shown in Figure 3-51. The **comments** in VHDL follow two dashes (--). Typing two successive dashes allows the designer to write information from that point to the end of the line. The information following the two successive dashes will be ignored by computer programs using this file, but can be read by any person trying to decipher the code.

The intermediate signal m is defined on line 13 following the keyword SIGNAL. The keyword BIT designates the type of the signal. Notice that a colon separates the signal name from its type designation. In the hardware description on line 16, the intermediate signal is assigned (connected to) a value

```
1   -- Intermediate variables in VHDL (Figure 3-49)
2   -- Digital Systems 10th ed
3   -- NS Widmer
4   -- MAY 23, 2005
5
6   ENTITY  fig3_51 IS
7   PORT( a, b, c   :IN BIT;      -- define inputs to block
8   y               :OUT BIT);    -- define block output
9   END fig3_51;
10
11  ARCHITECTURE ckt OF fig3_51 IS
12
13     SIGNAL m     :BIT;         -- name an intermediate signal
14
15  BEGIN
16        m <= a AND b;           -- generate buried product term
17        y <= m OR c;            -- generate sum on output
18  END ckt;
```

FIGURE 3-51 Intermediate signals in VHDL described in Figure 3-49.

(*m*<=*a* AND *b*;) and then *m* is used in the statement on line 17 to assign (connect) a value to *y* (*y*<=*m* OR *c*;). Remember that the assignment statements are concurrent and, thus, the order in which they are given does not matter. For human readability, it may seem more logical to assign values to intermediate signals before they are used in other assignment statements, as shown here.

REVIEW QUESTIONS

1. What is the designation used for intermediate signals?
2. Where are these signals declared?
3. Does it matter whether the m or y equation comes first?
4. What characters are used to comment a single line?

SUMMARY

1. Boolean algebra is a mathematical tool used in the analysis and design of digital circuits.
2. The basic Boolean operations are the OR, AND, and NOT operations.
3. An OR gate produces a HIGH output when any input is HIGH. An AND gate produces a HIGH output only when all inputs are HIGH. A NOT circuit (INVERTER) produces an output that is the opposite logic level compared to the input.
4. A NOR gate is the same as an OR gate with its output connected to an INVERTER. A NAND gate is the same as an AND gate with its output connected to an INVERTER.

5. Boolean theorems and rules can be used to simplify the expression of a logic circuit and can lead to a simpler way of implementing the circuit.
6. NAND gates can be used to implement any of the basic Boolean operations. NOR gates can be used likewise.
7. Either standard or alternate symbols can be used for each logic gate, depending on whether the output is to be active-HIGH or active-LOW.
8. The IEEE/ANSI standard for logic symbols uses rectangular symbols for each logic device, with special notations inside the rectangles to show how the outputs depend on the inputs.
9. Hardware description languages have become an important method of describing digital circuits.
10. HDL code should always contain comments that document its vital characteristics so a person reading it later can understand what it does.
11. Every HDL circuit description contains a definition of the inputs and outputs, followed by a section that describes the circuit's operation.
12. In addition to inputs and outputs, intermediate connections that are buried within the circuit can be defined. These intermediate connections are called nodes or signals.

IMPORTANT TERMS

- logic level
- Boolean algebra
- truth table
- OR operation
- OR gate
- AND operation
- AND gate
- NOT operation
- inversion (complementation)
- NOT circuit (INVERTER)
- NOR gate
- NAND gate
- Boolean theorems
- DeMorgan's theorems
- alternate logic symbols
- active logic levels
- active-HIGH
- active-LOW
- asserted
- unasserted
- IEEE/ANSI
- hardware description languages (HDLs)
- Altera hardware description language (AHDL)
- very high speed integrated circuit (VHSIC) hardware description language (VHDL)
- programmable logic devices (PLDs)
- concurrent
- compiler
- syntax
- mode
- type
- SUBDESIGN
- concurrent assignment statement
- ENTITY
- BIT
- ARCHITECTURE
- buried nodes (local signals)
- comments
- VARIABLE
- NODE

PROBLEMS

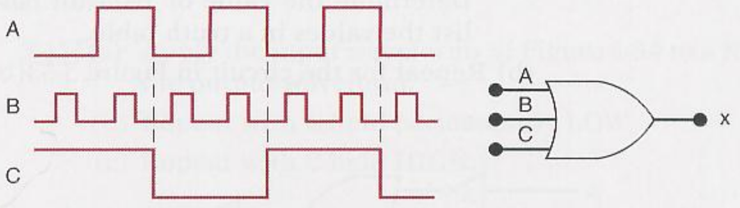
The color letters preceding some of the problems are used to indicate the nature or type of problem as follows:

B basic problem
T troubleshooting problem
D design or circuit-modification problem
N new concept or technique not covered in text
C challenging problem
H HDL problem

SECTION 3-3

B 3-1.* Draw the output waveform for the OR gate of Figure 3-52.

FIGURE 3-52



B 3-2. Suppose that the *A* input in Figure 3-52 is unintentionally shorted to ground (i.e., $A = 0$). Draw the resulting output waveform.

B 3-3.* Suppose that the *A* input in Figure 3-52 is unintentionally shorted to the +5 V supply line (i.e., $A = 1$). Draw the resulting output waveform.

C 3-4. Read the statements below concerning an OR gate. At first, they may appear to be valid, but after some thought you should realize that neither one is *always* true. Prove this by showing a specific example to refute each statement.

(a) If the output waveform from an OR gate is the same as the waveform at one of its inputs, the other input is being held permanently LOW.

(b) If the output waveform from an OR gate is always HIGH, one of its inputs is being held permanently HIGH.

B 3-5. How many different sets of input conditions will produce a HIGH output from a five-input OR gate?

SECTION 3-4

B 3-6. Change the OR gate in Figure 3-52 to an AND gate.

(a)* Draw the output waveform.

(b) Draw the output waveform if the *A* input is permanently shorted to ground.

(c) Draw the output waveform if *A* is permanently shorted to +5 V.

D 3-7.* Refer to Figure 3-4. Modify the circuit so that the alarm is to be activated only when the pressure and the temperature exceed their maximum limits at the same time.

B 3-8.* Change the OR gate in Figure 3-6 to an AND gate and draw the output waveform.

B 3-9. Suppose that you have an unknown two-input gate that is either an OR gate or an AND gate. What combination of input levels should you apply to the gate's inputs to determine which type of gate it is?

B 3-10. *True or false:* No matter how many inputs it has, an AND gate will produce a HIGH output for only one combination of input levels.

---

*Answers to problems marked with an asterisk can be found in the back of the text.

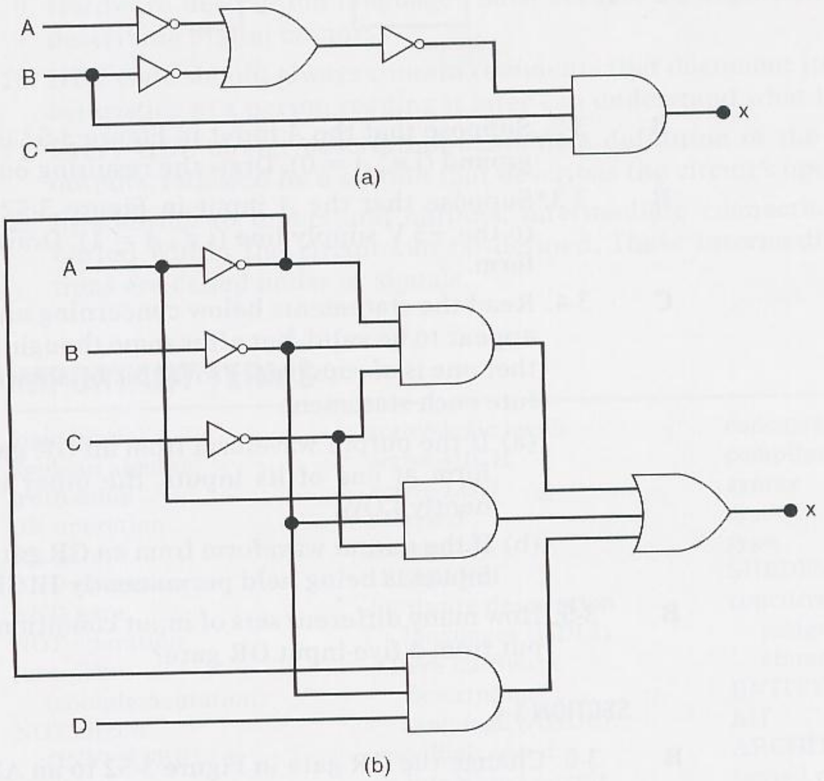
SECTIONS 3-5 TO 3-7

B 3-11. Apply the A waveform from Figure 3-23 to the input of an INVERTER. Draw the output waveform. Repeat for waveform B.

B 3-12. (a)* Write the Boolean expression for output x in Figure 3-53(a). Determine the value of x for all possible input conditions, and list the values in a truth table.

(b) Repeat for the circuit in Figure 3-53(b).

FIGURE 3-53



(a)

(b)

B 3-13.* Create a complete analysis table for the circuit of Figure 3-15(b) by finding the logic levels present at each gate output for each of the 32 possible input combinations.

B 3-14. (a)* Change each OR to an AND, and each AND to an OR, in Figure 3-15(b). Then write the expression for the output.

(b) Complete an analysis table.

B 3-15. Create a complete analysis table for the circuit of Figure 3-16 by finding the logic levels present at each gate output for each of the 16 possible combinations of input levels.

SECTION 3-8

B 3-16. For each of the following expressions, construct the corresponding logic circuit, using AND and OR gates and INVERTERs.

(a)* $x = \overline{AB(C + D)}$

(b)* $z = \overline{A + B + \overline{C}D\overline{E}}) + \overline{B}C\overline{D}$

(c) $y = (\overline{M + N} + \overline{P}Q)$

(d) $x = \overline{W + P\overline{Q}}$

(e) $z = MN(P + \overline{N})$

(f) $x = (A + B)(\overline{A} + \overline{B})$

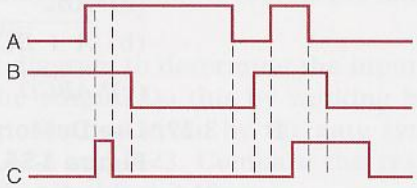
SECTION 3-9

B 3-17.* (a) Apply the input waveforms of Figure 3-54 to a NOR gate, and draw the output waveform.

(b) Repeat with C held permanently LOW.

(c) Repeat with C held HIGH.

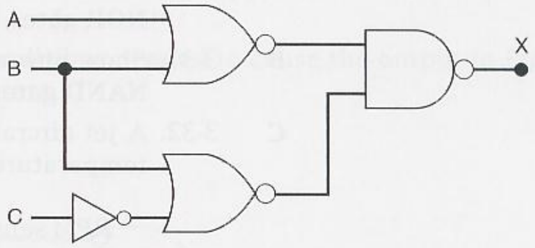
FIGURE 3-54



B 3-18. Repeat Problem 3-17 for a NAND gate.

C 3-19.* Write the expression for the output of Figure 3-55, and use it to determine the complete truth table. Then apply the waveforms of Figure 3-54 to the circuit inputs, and draw the resulting output waveform.

FIGURE 3-55



B 3-20. Determine the truth table for the circuit of Figure 3-24.

B 3-21. Modify the circuits that were constructed in Problem 3-16 so that NAND gates and NOR gates are used wherever appropriate.

SECTION 3-10

C 3-22. Prove theorems (15a) and (15b) by trying all possible cases.

B 3-23.* DRILL QUESTION

Complete each expression.

(a) $A + 1 =$ __________

(b) $A \cdot A =$ __________

(c) $B \cdot \overline{B} =$ __________

(d) $C + C =$ __________

(e) $x \cdot 0 =$ __________

(f) $D \cdot 1 =$ __________

(g) $D + 0 =$ __________

(h) $C + \overline{C} =$ __________

(i) $G + GF =$ __________

(j) $y + \overline{w}y =$ __________

C 3-24. (a)* Simplify the following expression using theorems (13b), (3), and (4):

$$x = (M + N)(\overline{M} + P)(\overline{N} + \overline{P})$$

(b) Simplify the following expression using theorems (13a), (8), and (6):

$$z = \overline{A}B\overline{C} + AB\overline{C} + B\overline{C}D$$

SECTIONS 3-11 AND 3-12

C 3-25. Prove DeMorgan's theorems by trying all possible cases.

B 3-26. Simplify each of the following expressions using DeMorgan's theorems.

(a)* $\overline{\overline{A}B\overline{C}}$
(b) $\overline{\overline{A} + \overline{BC}}$
(c)* $\overline{AB\overline{CD}}$
(d) $\overline{A + \overline{B}}$
(e)* $\overline{\overline{AB}}$
(f) $\overline{\overline{A} + \overline{C} + \overline{D}}$
(g)* $\overline{A(\overline{B + \overline{C}})D}$
(h) $\overline{(M + \overline{N})(\overline{M} + N)}$
(i) $\overline{\overline{\overline{AB}C}D}$

B 3-27.* Use DeMorgan's theorems to simplify the expression for the output of Figure 3-55.

C 3-28. Convert the circuit of Figure 3-53(b) to one using only NAND gates. Then write the output expression for the new circuit, simplify it using DeMorgan's theorems, and compare it with the expression for the original circuit.

C 3-29. Convert the circuit of Figure 3-53(a) to one using only NOR gates. Then write the expression for the new circuit, simplify it using DeMorgan's theorems, and compare it with the expression for the original circuit.

B 3-30. Show how a two-input NAND gate can be constructed from two-input NOR gates.

B 3-31. Show how a two-input NOR gate can be constructed from two-input NAND gates.

C 3-32. A jet aircraft employs a system for monitoring the rpm, pressure, and temperature values of its engines using sensors that operate as follows:

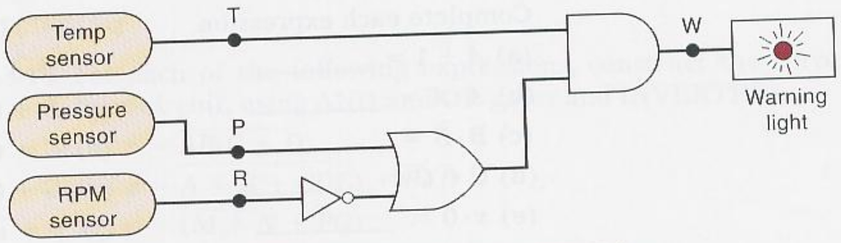
RPM sensor output = 0 only when speed < 4800 rpm
P sensor output = 0 only when pressure < 220 psi
T sensor output = 0 only when temperature < 200°F

Figure 3-56 shows the logic circuit that controls a cockpit warning light for certain combinations of engine conditions. Assume that a HIGH at output *W* activates the warning light.

(a)* Determine what engine conditions will give a warning to the pilot.

(b) Change this circuit to one using all NAND gates.

FIGURE 3-56



SECTIONS 3-13 AND 3-14

B 3-33. For each statement below, draw the appropriate logic-gate symbol—standard or alternate—for the given operation.

(a) A HIGH output occurs only when all three inputs are LOW.

(b) A LOW output occurs when any of the four inputs is LOW.

(c) A LOW output occurs only when all eight inputs are HIGH.

B 3-34. Draw the standard representations for each of the basic logic gates. Then draw the alternate representations.

C 3-35. The circuit of Figure 3-55 is supposed to be a simple digital combination lock whose output will generate an active-LOW $\overline{UNLOCK}$ signal for only one combination of inputs.

(a)* Modify the circuit diagram so that it represents more effectively the circuit operation.

(b) Use the new circuit diagram to determine the input combination that will activate the output. Do this by working back from the output using the information given by the gate symbols, as was done in Examples 3-22 and 3-23. Compare the results with the truth table obtained in Problem 3-19.

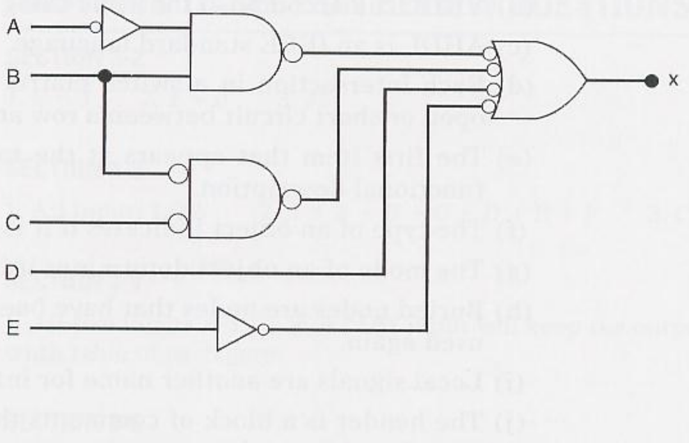
C 3-36. (a) Determine the input conditions needed to activate output Z in Figure 3-37(b). Do this by working back from the output, as was done in Examples 3-22 and 3-23.

(b) Assume that it is the LOW state of Z that is to activate the alarm. Change the circuit diagram to reflect this, and then use the revised diagram to determine the input conditions needed to activate the alarm.

D 3-37. Modify the circuit of Figure 3-40 so that $A_1 = 0$ is needed to produce $DRIVE = 1$ instead of $A_1 = 1$.

B 3-38.* Determine the input conditions needed to cause the output in Figure 3-57 to go to its active state.

FIGURE 3-57

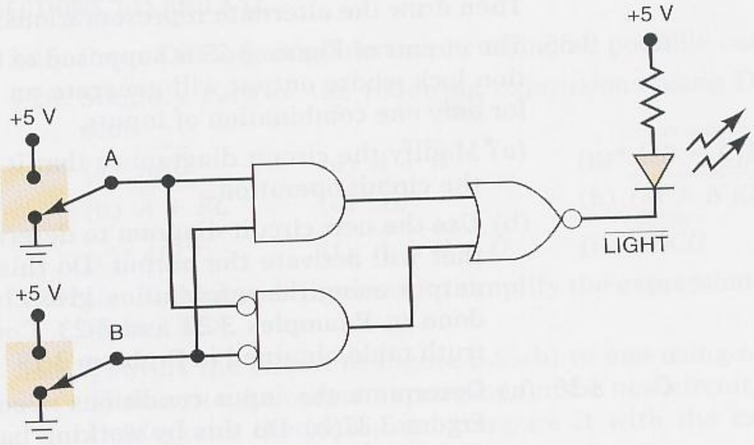


B 3-39.* What is the asserted state for the output of Figure 3-57? For the output of Figure 3-36(c)?

B 3-40. Use the results of Problem 3-38 to obtain the complete truth table for the circuit of Figure 3-57.

N 3-41.* Figure 3-58 shows an application of logic gates that simulates a two-way switch like the ones used in our homes to turn a light on or off from two different switches. Here the light is an LED that will be ON (conducting) when the NOR gate output is LOW. Note that this output is labeled *LIGHT* to indicate that it is active-LOW. Determine the input conditions needed to turn on the LED. Then verify that the circuit operates as a two-way switch using switches *A* and *B*. (In Chapter 4, you will learn how to design circuits like this one to produce a given relationship between inputs and outputs.)

FIGURE 3-58



SECTION 3-15

B 3-42. Redraw the circuits of (a)* Figure 3-57 and (b) Figure 3-58 using the IEEE/ANSI symbols.

SECTION 3-17

HDL DRILL QUESTIONS

H 3-43.* *True or false:*

(a) VHDL is a computer programming language.

(b) VHDL can accomplish the same thing as AHDL.

(c) AHDL is an IEEE standard language.

(d) Each intersection in a switch matrix can be programmed as an open or short circuit between a row and column wire.

(e) The first item that appears at the top of an HDL listing is the functional description.

(f) The type of an object indicates if it is an input or an output.

(g) The mode of an object determines if it is an input or an output.

(h) Buried nodes are nodes that have been deleted and will never be used again.

(i) Local signals are another name for intermediate variables.

(j) The header is a block of comments that document vital information about the project.

SECTION 3-18

B 3-44. Redraw the programmable connection matrix from Figure 3-44. Label the output signals (horizontal lines) from the connection matrix (from top row to bottom row) as follows: AAABADHE. Draw an X in the appropriate intersections to short-circuit a row to a column and create these connections to the logic circuit.

H 3-45.*Write the HDL code in the language of your choice that will produce the following output functions:

$$X = A + B$$
$$Y = AB$$
$$Z = A + B + C$$

H 3-46. Write the HDL code in the language of your choice that will implement the logic circuit of Figure 3-39.

(a) Use a single Boolean equation.

(b) Use the intermediate variables V, W, X, and Y.

MICROCOMPUTER APPLICATION

C 3-47.*Refer to Figure 3-40 in Example 3-23. Inputs A_7 through A_0 are *address* inputs that are supplied to this circuit from outputs of the microprocessor chip in a microcomputer. The eight-bit address code A_7 to A_0 selects which device the microprocessor wants to activate. In Example 3-23, the required address code to activate the disk drive was A_7 through $A_0 = 11111110_2 = \text{FE}_{16}$.

Modify the circuit so that the microprocessor must supply an address code of 4A_{16} to activate the disk drive.

CHALLENGING EXERCISES

C 3-48. Show how $x = AB\overline{C}$ can be implemented with one two-input NOR and one two-input NAND gate.

C 3-49.*Implement $y = ABCD$ using only two-input NAND gates.

ANSWERS TO SECTION REVIEW QUESTIONS

SECTION 3-2

1. $x = 1$ 2. $x = 0$ 3. 32

SECTION 3-3

1. All inputs LOW 2. $x = A + B + C + D + E + F$ 3. Constant HIGH

SECTION 3-4

1. All five inputs = 1 2. A LOW input will keep the output LOW. 3. False; see truth table of each gate.

SECTION 3-5

1. Output of second INVERTER will be the same as input A. 2. y will be LOW only for $A = B = 1$.

SECTION 3-6

1. $x = \overline{A} + B + C + \overline{AD}$ 2. $x = D(\overline{AB + C}) + E$

SECTION 3-7

1. $x = 1$ 2. $x = 1$ 3. $x = 1$ for both.

SECTION 3-8

1. See Figure 3-15(a). 2. See Figure 3-17(b). 3. See Figure 3-15(b).

SECTION 3-9

1. All inputs LOW. 2. $x = 0$ 3. $x = \overline{A + B + \overline{CD}}$

SECTION 3-10

1. $y = A\overline{C}$ 2. $y = \overline{A}\,\overline{B}\,\overline{D}$ 3. $y = \overline{A}D + BD$

SECTION 3-11

1. $z = \overline{AB} + C$ 2. $y = (\overline{R} + S + \overline{T})Q$ 3. Same as Figure 3-28 except NAND is replaced by NOR. 4. $y = \overline{A}B(C + \overline{D})$

SECTION 3-12

1. Three. 2. NOR circuit is more efficient because it can be implemented with one 74LS02 IC. 3. $x = \overline{(\overline{AB})\,(\overline{CD})} = \overline{\overline{AB}} + \overline{\overline{(CD)}} + AB + CD$

SECTION 3-13

1. Output goes LOW when any input is HIGH. 2. Output goes HIGH only when all inputs are LOW. 3. Output goes LOW when any input is LOW. 4. Output goes HIGH only when all inputs are HIGH.

SECTION 3-14

1. Z will go HIGH when $A = B = 0$ and $C = D = 1$. 2. Z will go LOW when $A = B = 0$, $E = 1$, and either C or D or both are 0. 3. Two 4. Two 5. LOW 6. $A = B = 0$, $C = D = 1$ 7. $\overline{W}$

SECTION 3-15

1. See Figure 3-41. 2. Rectangle with & inside, and triangles on inputs.

SECTION 3-16

1. Boolean equation, truth table, logic diagram, timing diagram, language.

SECTION 3-17

1. Hardware description language 2. To describe a digital circuit and its operation. 3. To give a computer a sequential list of tasks. 4. HDL describes concurrent hardware circuits; computer instructions execute one at a time.

SECTION 3-18

1. Programmable logic device 2. By making and breaking connections in a switching matrix 3. It translates HDL code into a pattern of bits to configure the switching matix.

SECTION 3-19

AHDL

1. The input and output definitions. 2. The description of how it operates.

VHDL

1. To give a name to the circuit and define its inputs and outputs. 2. The ARCHITECTURE description. 3. <=

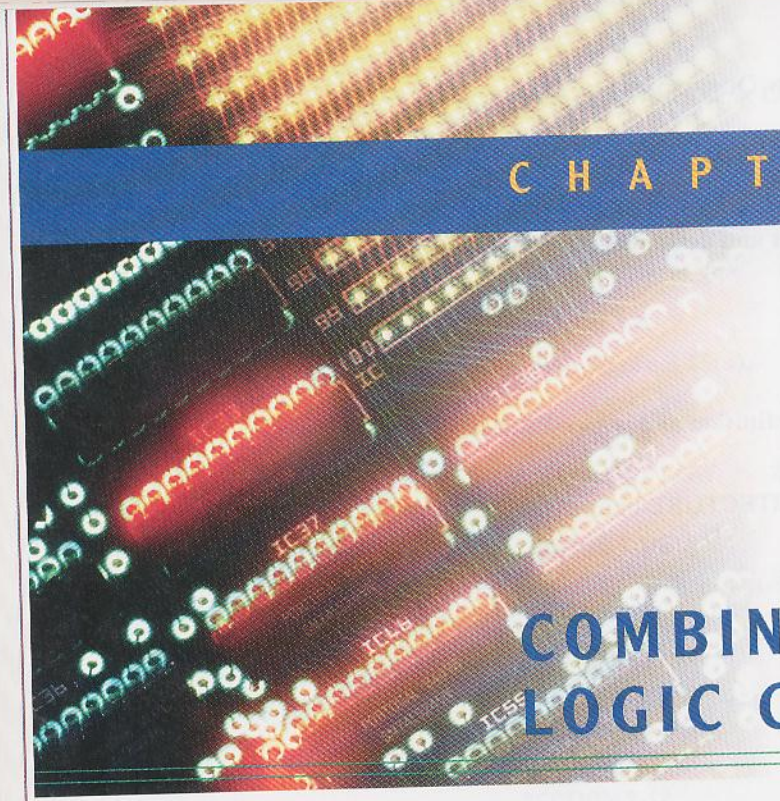
SECTION 3-20

AHDL

1. NODE 2. After the I/O definition and before BEGIN. 3. No 4. % 5. --

VHDL

1. SIGNAL 2. Inside ARCHITECTURE before BEGIN. 3. No 4. --



CHAPTER 4

COMBINATIONAL LOGIC CIRCUITS

OUTLINE

4-1 Sum-of-Products Form
4-2 Simplifying Logic Circuits
4-3 Algebraic Simplification
4-4 Designing Combinational Logic Circuits
4-5 Karnaugh Map Method
4-6 Exclusive-OR and Exclusive-NOR Circuits
4-7 Parity Generator and Checker
4-8 Enable/Disable Circuits
4-9 Basic Characteristics of Digital ICs
4-10 Troubleshooting Digital Systems
4-11 Internal Digital IC Faults
4-12 External Faults
4-13 Troubleshooting Case Study
4-14 Programmable Logic Devices
4-15 Representing Data in HDL
4-16 Truth Tables Using HDL
4-17 Decision Control Structures in HDL

OBJECTIVES

Upon completion of this chapter, you will be able to:

- Convert a logic expression into a sum-of-products expression.
- Perform the necessary steps to reduce a sum-of-products expression to its simplest form.
- Use Boolean algebra and the Karnaugh map as tools to simplify and design logic circuits.
- Explain the operation of both exclusive-OR and exclusive-NOR circuits.
- Design simple logic circuits without the help of a truth table.
- Implement enable circuits.
- Cite the basic characteristics of TTL and CMOS digital ICs.
- Use the basic troubleshooting rules of digital systems.
- Deduce from observed results the faults of malfunctioning combinational logic circuits.
- Describe the fundamental idea of programmable logic devices (PLDs).
- Outline the steps involved in programming a PLD to perform a simple combinational logic function.
- Go to the Altera user manuals to acquire the information needed to do a simple programming experiment in the lab.
- Describe hierarchical design methods.
- Identify proper data types for single-bit, bit array, and numeric value variables.
- Describe logic circuits using HDL control structures IF/ELSE, IF/ELSIF, and CASE.
- Select the appropriate control structure for a given problem.

INTRODUCTION

In Chapter 3, we studied the operation of all the basic logic gates, and we used Boolean algebra to describe and analyze circuits that were made up of combinations of logic gates. These circuits can be classified as *combinational* logic circuits because, at any time, the logic level at the output depends on the combination of logic levels present at the inputs. A combinational circuit has no *memory* characteristic, so its output depends *only* on the current value of its inputs.

In this chapter, we will continue our study of combinational circuits. To start, we will go further into the simplification of logic circuits. Two methods will be used: one uses Boolean algebra theorems; the other uses a *mapping* technique. In addition, we will study simple techniques for designing

combinational logic circuits to satisfy a given set of requirements. A complete study of logic-circuit design is not one of our objectives, but the methods we introduce will provide a good introduction to logic design.

A good portion of the chapter is devoted to the troubleshooting of combinational circuits. This first exposure to troubleshooting should begin to develop the type of analytical skills needed for successful troubleshooting. To make this material as practical as possible, we will first present some of the basic characteristics of logic-gate ICs in the TTL and CMOS logic families along with a description of the most common types of faults encountered in digital IC circuits.

In the last sections of this chapter, we will extend our knowledge of programmable logic devices and hardware description languages. The concept of programmable hardware connections will be reinforced, and we will provide more details regarding the role of the development system. You will learn the steps followed in the design and development of digital systems today. Enough information will be provided to allow you to choose the correct types of data objects for use in simple projects to be presented later in this text. Finally, several control structures will be explained, along with some instruction regarding their appropriate use.

4-1 SUM-OF-PRODUCTS FORM

The methods of logic-circuit simplification and design that we will study require the logic expression to be in a **sum-of-products (SOP)** form. Some examples of this form are:

1. $ABC + \overline{A}B\overline{C}$
2. $AB + \overline{A}B\overline{C} + \overline{C}\,\overline{D} + D$
3. $\overline{A}B + C\overline{D} + EF + GK + H\overline{L}$

Each of these sum-of-products expressions consists of two or more AND terms (products) that are ORed together. Each AND term consists of one or more variables *individually* appearing in either complemented or uncomplemented form. For example, in the sum-of-products expression $ABC + \overline{A}B\overline{C}$, the first AND product contains the variables A, B, and C in their uncomplemented (not inverted) form. The second AND term contains A and C in their complemented (inverted) form. Note that in a sum-of-products expression, one inversion sign *cannot* cover more than one variable in a term (e.g., we cannot have $\overline{ABC}$ or $\overline{RST}$).

Product-of-Sums

Another general form for logic expressions is sometimes used in logic-circuit design. Called the **product-of-sums (POS)** form, it consists of two or more OR terms (sums) that are ANDed together. Each OR term contains one or more variables in complemented or uncomplemented form. Here are some product-of-sum expressions:

1. $(A + \overline{B} + C)(A + C)$
2. $(A + \overline{B})(\overline{C} + D)F$
3. $(A + C)(B + \overline{D})(\overline{B} + C)(A + \overline{D} + \overline{E})$

The methods of circuit simplification and design that we will be using are based on the sum-of-products (SOP) form, so we will not be doing much with the product-of-sums (POS) form. It will, however, occur from time to time in some logic circuits that have a particular structure.

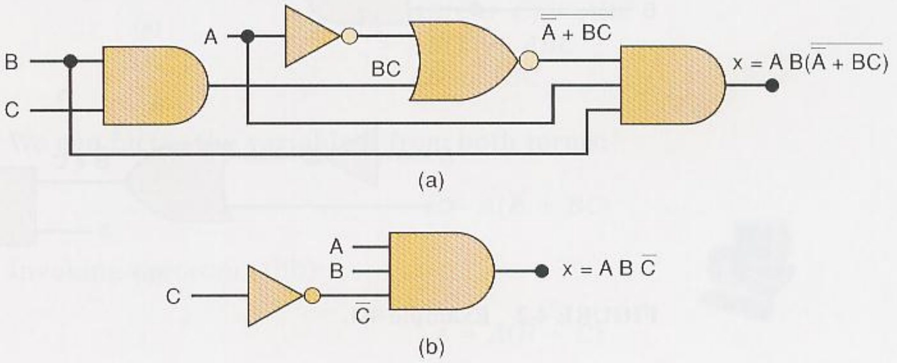
REVIEW QUESTIONS

1. Which of the following expressions is in SOP form?
   - (a) $AB + CD + E$
   - (b) $AB(C + D)$
   - (c) $(A + B)(C + D + F)$
   - (d) $\overline{MN} + PQ$
2. Repeat question 1 for the POS form.

4-2 SIMPLIFYING LOGIC CIRCUITS

Once the expression for a logic circuit has been obtained, we may be able to reduce it to a simpler form containing fewer terms or fewer variables in one or more terms. The new expression can then be used to implement a circuit that is equivalent to the original circuit but that contains fewer gates and connections.

To illustrate, the circuit of Figure 4-1(a) can be simplified to produce the circuit of Figure 4-1(b). Both circuits perform the same logic, so it should be obvious that the simpler circuit is more desirable because it contains fewer gates and will therefore be smaller and cheaper than the original. Furthermore, the circuit reliability will improve because there are fewer interconnections that can be potential circuit faults.



(a)

(b)

FIGURE 4-1 It is often possible to simplify a logic circuit such as that in part (a) to produce a more efficient implementation, shown in (b).

In subsequent sections, we will study two methods for simplifying logic circuits. One method will utilize the Boolean algebra theorems and, as we shall see, is greatly dependent on inspiration and experience. The other method (Karnaugh mapping) is a systematic, step-by-step approach. Some instructors may wish to skip over this latter method because it is somewhat mechanical and probably does not contribute to a better understanding of Boolean algebra. This can be done without affecting the continuity or clarity of the rest of the text.

4-3 ALGEBRAIC SIMPLIFICATION

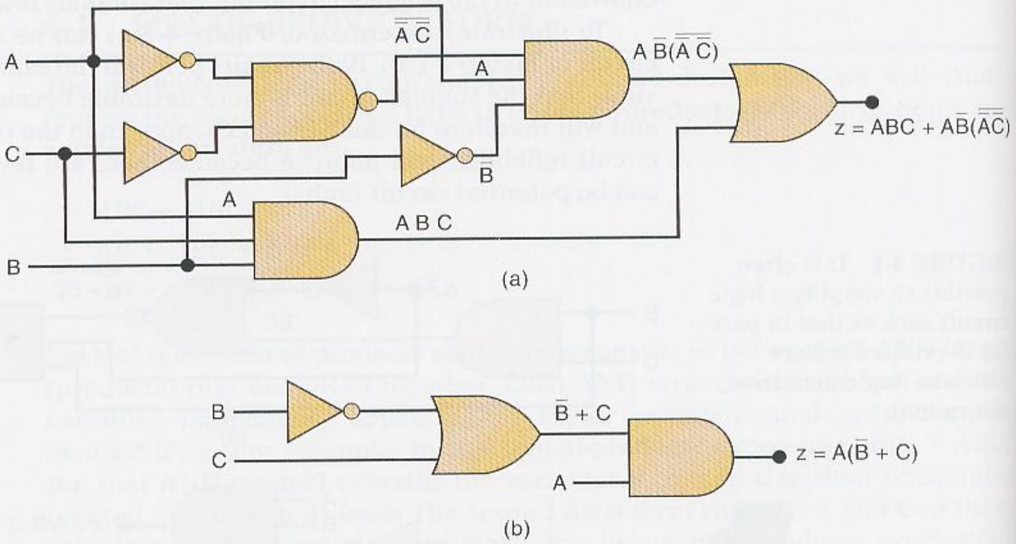
We can use the Boolean algebra theorems that we studied in Chapter 3 to help us simplify the expression for a logic circuit. Unfortunately, it is not always obvious which theorems should be applied to produce the simplest result. Furthermore, there is no easy way to tell whether the simplified expression is in its simplest form or whether it could have been simplified further. Thus, algebraic simplification often becomes a process of trial and error. With experience, however, one can become adept at obtaining reasonably good results.

The examples that follow will illustrate many of the ways in which the Boolean theorems can be applied in trying to simplify an expression. You should notice that these examples contain two essential steps:

1. The original expression is put into SOP form by repeated application of DeMorgan's theorems and multiplication of terms.
2. Once the original expression is in SOP form, the product terms are checked for common factors, and factoring is performed wherever possible. The factoring should result in the elimination of one or more terms.

EXAMPLE 4-1

Simplify the logic circuit shown in Figure 4-2(a).



A
C
B
$\overline{\bar{A}\,\bar{C}}$
A
$A\,\bar{B}(\overline{\bar{A}\,\bar{C}})$
$z = ABC + A\bar{B}(\overline{\bar{A}\bar{C}})$
$\bar{B}$
A
A B C
(a)
B
C
$\bar{B} + C$
A
$z = A(\bar{B} + C)$
(b)

FIGURE 4-2 Example 4-1.

Solution

The first step is to determine the expression for the output using the method presented in Section 3-6. The result is

$$z = ABC + A\bar{B}\cdot(\overline{\bar{A}\,\bar{C}})$$

Once the expression is determined, it is usually a good idea to break down all large inverter signs using DeMorgan's theorems and then multiply out all terms.

$$\begin{aligned} z &= ABC + A\bar{B}(\bar{\bar{A}} + \bar{\bar{C}}) && \text{[theorem (17)]} \\ &= ABC + A\bar{B}(A + C) && \text{[cancel double inversions]} \\ &= ABC + A\bar{B}A + A\bar{B}C && \text{[multiply out]} \\ &= ABC + A\bar{B} + A\bar{B}C && [A \cdot A = A] \end{aligned}$$

With the expression now in SOP form, we should look for common variables among the various terms with the intention of factoring. The first and third terms above have AC in common, which can be factored out:

$$z = AC(B + \overline{B}) + A\overline{B}$$

Since $B + \overline{B} = 1$, then

$$\begin{aligned} z &= AC(1) + A\overline{B} \\ &= AC + A\overline{B} \end{aligned}$$

We can now factor out A, which results in

$$z = A(C + \overline{B})$$

This result can be simplified no further. Its circuit implementation is shown in Figure 4-2(b). It is obvious that the circuit in Figure 4-2(b) is a great deal simpler than the original circuit in Figure 4-2(a).

EXAMPLE 4-2

Simplify the expression $z = A\overline{B}\,\overline{C} + A\overline{B}C + ABC$.

Solution

The expression is already in SOP form.

Method 1: The first two terms in the expression have the product $A\overline{B}$ in common. Thus,

$$\begin{aligned} z &= A\overline{B}(\overline{C} + C) + ABC \\ &= A\overline{B}(1) + ABC \\ &= A\overline{B} + ABC \end{aligned}$$

We can factor the variable A from both terms:

$$z = A(\overline{B} + BC)$$

Invoking theorem (15b):

$$z = A(\overline{B} + C)$$

Method 2: The original expression is $z = A\overline{B}\,\overline{C} + A\overline{B}C + ABC$. The first two terms have $A\overline{B}$ in common. The last two terms have AC in common. How do we know whether to factor $A\overline{B}$ from the first two terms or AC from the last two terms? Actually, we can do both by using the $A\overline{B}C$ term *twice*. In other words, we can rewrite the expression as:

$$z = A\overline{B}\,\overline{C} + A\overline{B}C + A\overline{B}C + ABC$$

where we have added an extra term $A\overline{B}C$. This is valid and will not change the value of the expression because $A\overline{B}C + A\overline{B}C = A\overline{B}C$ [theorem (7)]. Now we can factor $A\overline{B}$ from the first two terms and AC from the last two terms:

$$\begin{aligned} z &= A\overline{B}(C + \overline{C}) + AC(\overline{B} + B) \\ &= A\overline{B}\cdot 1 + AC\cdot 1 \\ &= A\overline{B} + AC = A(\overline{B} + C) \end{aligned}$$

Of course, this is the same result obtained with method 1. This trick of using the same term twice can always be used. In fact, the same term can be used more than twice if necessary.

EXAMPLE 4-3

Simplify $z = \overline{A}C(\overline{\overline{A}BD}) + \overline{A}B\overline{C}\,\overline{D} + A\overline{B}C$.

Solution

First, use DeMorgan's theorem on the first term:

$$z = \overline{A}C(A + \overline{B} + \overline{D}) + \overline{A}B\overline{C}\,\overline{D} + A\overline{B}C \qquad \textbf{(step 1)}$$

Multiplying out yields

$$z = \overline{A}CA + \overline{A}C\overline{B} + \overline{A}C\overline{D} + \overline{A}B\overline{C}\,\overline{D} + A\overline{B}C \qquad (2)$$

Because $\overline{A} \cdot A = 0$, the first term is eliminated:

$$z = \overline{A}\,\overline{B}C + \overline{A}C\overline{D} + \overline{A}B\overline{C}\,\overline{D} + A\overline{B}C \qquad (3)$$

This is the desired SOP form. Now we must look for common factors among the various product terms. The idea is to check for the largest common factor between any two or more product terms. For example, the first and last terms have the common factor $\overline{B}C$, and the second and third terms share the common factor $\overline{A}\,\overline{D}$. We can factor these out as follows:

$$z = \overline{B}C(\overline{A} + A) + \overline{A}\,\overline{D}(C + B\overline{C}) \qquad (4)$$

Now, because $\overline{A} + A = 1$, and $C + B\overline{C} = C + B$ [theorem (15a)], we have

$$z = \overline{B}C + \overline{A}\,\overline{D}(B + C) \qquad (5)$$

This same result could have been reached with other choices for the factoring. For example, we could have factored C from the first, second, and fourth product terms in step 3 to obtain

$$z = C(\overline{A}\,\overline{B} + \overline{A}\,\overline{D} + A\overline{B}) + \overline{A}B\overline{C}\,\overline{D}$$

The expression inside the parentheses can be factored further:

$$z = C(\overline{B}[\overline{A} + A] + \overline{A}\,\overline{D}) + \overline{A}B\overline{C}\,\overline{D}$$

Because $\overline{A} + A = 1$, this becomes

$$z = C(\overline{B} + \overline{A}\,\overline{D}) + \overline{A}B\overline{C}\,\overline{D}$$

Multiplying out yields

$$z = \overline{B}C + \overline{A}C\,\overline{D} + \overline{A}B\overline{C}\,\overline{D}$$

Now we can factor $\overline{A}\,\overline{D}$ from the second and third terms to get

$$z = \overline{B}C + \overline{A}\,\overline{D}(C + B\overline{C})$$

If we use theorem (15a), the expression in parentheses becomes $B + C$. Thus, we finally have

$$z = \overline{B}C + \overline{A}\,\overline{D}(B + C)$$

This is the same result that we obtained earlier, but it took us many more steps. This illustrates why you should look for the largest common factors: it will generally lead to the final expression in the fewest steps.

Example 4-3 illustrates the frustration often encountered in Boolean simplification. Because we have arrived at the same equation (which appears irreducible) by two different methods, it might seem reasonable to conclude that this final equation is the simplest form. In fact, the simplest form of this equation is

$$z = \overline{A}B\overline{D} + \overline{B}C$$

But there is no apparent way to reduce step (5) to reach this simpler version. In this case, we missed an operation earlier in the process that could have led to the simpler form. The question is, "How could we have known that we missed a step?" Later in this chapter, we will examine a mapping technique that will always lead to the simplest SOP form.

EXAMPLE 4-4

Simplify the expression $x = (\overline{A} + B)(A + B + D)\overline{D}$.

Solution

The expression can be put into sum-of-products form by multiplying out all the terms. The result is

$$x = \overline{A}A\overline{D} + \overline{A}B\overline{D} + \overline{A}D\overline{D} + BA\overline{D} + BB\overline{D} + BD\overline{D}$$

The first term can be eliminated because $\overline{A}A = 0$. Likewise, the third and sixth terms can be eliminated because $D\overline{D} = 0$. The fifth term can be simplified to $B\overline{D}$ because $BB = B$. This gives us

$$x = \overline{A}B\overline{D} + AB\overline{D} + B\overline{D}$$

We can factor $B\overline{D}$ from each term to obtain

$$x = B\overline{D}(\overline{A} + A + 1)$$

Clearly, the term inside the parentheses is always 1, so we finally have

$$x = B\overline{D}$$

EXAMPLE 4-5

Simplify the circuit of Figure 4-3(a).

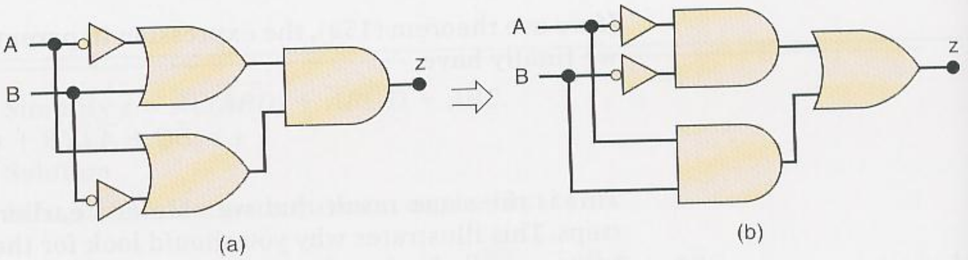


FIGURE 4-3 Example 4-5.

Solution

The expression for output z is

$$z = (\overline{A} + B)(A + \overline{B})$$

Multiplying out to get the sum-of-products form, we obtain

$$z = \overline{A}A + \overline{A}\,\overline{B} + BA + B\overline{B}$$

We can eliminate $\overline{A}A = 0$ and $B\overline{B} = 0$ to end up with

$$z = \overline{A}\,\overline{B} + AB$$

This expression is implemented in Figure 4-3(b), and if we compare it with the original circuit, we see that both circuits contain the same number of gates and connections. In this case, the simplification process produced an equivalent, but not simpler, circuit.

EXAMPLE 4-6

Simplify $x = A\overline{B}C + \overline{A}BD + \overline{C}\,\overline{D}$.

Solution

You can try, but you will not be able to simplify this expression any further.

REVIEW QUESTIONS

1. State which of the following expressions are *not* in the sum-of-products form:
   (a) $RS\overline{T} + \overline{R}S\overline{T} + \overline{T}$
   (b) $A\overline{DC} + \overline{A}DC$
   (c) $MN\overline{P} + (M + \overline{N})P$
   (d) $AB + \overline{A}B\overline{C} + A\overline{B}\,\overline{C}D$
2. Simplify the circuit in Figure 4-1(a) to arrive at the circuit of Figure 4-1(b).
3. Change each AND gate in Figure 4-1(a) to a NAND gate. Determine the new expression for x and simplify it.

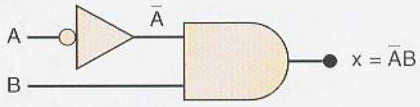
4-4 DESIGNING COMBINATIONAL LOGIC CIRCUITS

When the desired output level of a logic circuit is given for all possible input conditions, the results can be conveniently displayed in a truth table. The Boolean expression for the required circuit can then be derived from the truth table. For example, consider Figure 4-4(a), where a truth table is shown for a circuit that has two inputs, A and B, and output x. The table shows that output x is to be at the 1 level *only* for the case where $A = 0$ and $B = 1$. It now remains to determine what logic circuit will produce this desired operation. It should be apparent that one possible solution is that shown in Figure 4-4(b). Here an AND gate is used with inputs $\overline{A}$ and B, so that $x = \overline{A} \cdot B$. Obviously x will be 1 *only if* both inputs to the AND gate are 1, namely, $\overline{A} = 1$ (which means that $A = 0$) and $B = 1$. For all other values of A and B, the output x will be 0.

FIGURE 4-4 Circuit that produces a 1 output only for the $A = 0, B = 1$ condition.

A	B	x
0	0	0
0	1	1
1	0	0
1	1	0

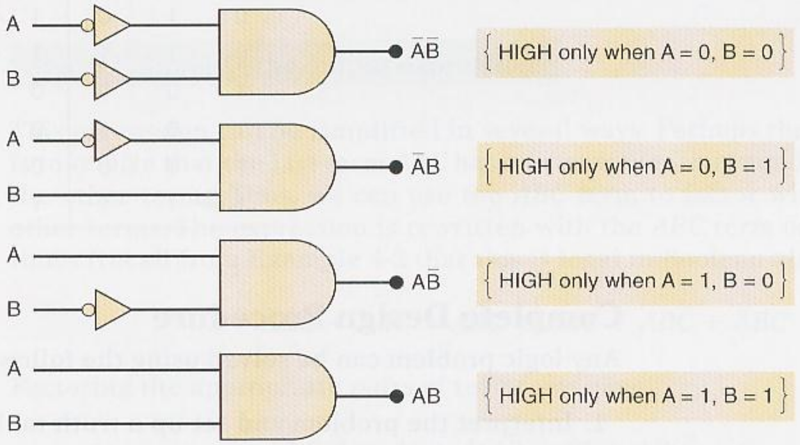
(a)



(b)

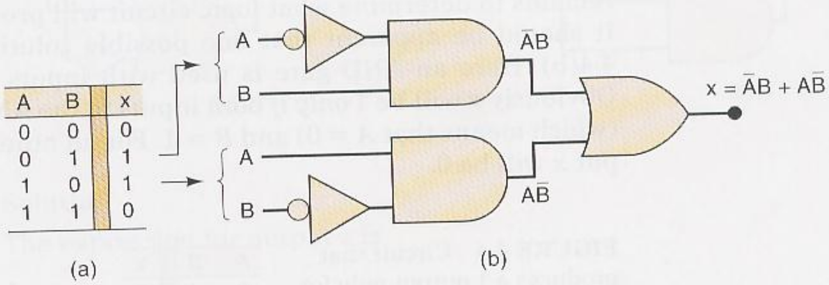
A similar approach can be used for the other input conditions. For instance, if x were to be high only for the $A = 1, B = 0$ condition, the resulting circuit would be an AND gate with inputs A and $\overline{B}$. In other words, for any of the four possible input conditions, we can generate a high x output by using an AND gate with appropriate inputs to generate the required AND product. The four different cases are shown in Figure 4-5. Each of the AND gates shown generates an output that is 1 *only* for one given input condition and the output is 0 for all other conditions. It should be noted that the AND inputs are inverted or not inverted depending on the values that the variables have for the given condition. If the variable is 0 for the given condition, it is inverted before entering the AND gate.

FIGURE 4-5 An AND gate with appropriate inputs can be used to produce a 1 output for a specific set of input levels.


A
B
$\overline{A}\overline{B}$
{ HIGH only when A = 0, B = 0 }
A
B
$\overline{A}B$
{ HIGH only when A = 0, B = 1 }
A
B
$A\overline{B}$
{ HIGH only when A = 1, B = 0 }
A
B
AB
{ HIGH only when A = 1, B = 1 }

Let us now consider the case shown in Figure 4-6(a), where we have a truth table that indicates that the output x is to be 1 for two different cases: $A = 0, B = 1$ and $A = 1, B = 0$. How can this be implemented? We know that the AND term $\overline{A}\cdot B$ will generate a 1 only for the $A = 0, B = 1$ condition, and the AND term $A\cdot\overline{B}$ will generate a 1 for the $A = 1, B = 0$ condition. Because x must be HIGH for *either* condition, it should be clear that these terms should be ORed together to produce the desired output, x. This implementation is shown in Figure 4-6(b), where the resulting expression for the output is $x = \overline{A}B + A\overline{B}$.

FIGURE 4-6 Each set of input conditions that is to produce a HIGH output is implemented by a separate AND gate. The AND outputs are ORed to produce the final output.



In this example, an AND term is generated for each case in the table where the output x is to be a 1. The AND gate outputs are then ORed together to produce the total output x, which will be 1 when either AND term is 1. This same procedure can be extended to examples with more than two inputs. Consider the truth table for a three-input circuit (Table 4-1). Here there are three cases where the output x is to be 1. The required AND term for each of these cases is shown. Again, note that for each case where a variable is 0, it appears inverted in the AND term. The sum-of-products expression for x is obtained by ORing the three AND terms.

$$x = \overline{A}B\overline{C} + \overline{A}BC + ABC$$

TABLE 4-1

A	*B*	*C*	*x*	
0	0	0	0	
0	0	1	0	
0	1	0	1	→ $\overline{A}B\overline{C}$
0	1	1	1	→ $\overline{A}BC$
1	0	0	0	
1	0	1	0	
1	1	0	0	
1	1	1	1	→ ABC

Complete Design Procedure

Any logic problem can be solved using the following step-by-step procedure.

1. Interpret the problem and set up a truth table to describe its operation.
2. Write the AND (product) term for each case where the output is 1.
3. Write the sum-of-products (SOP) expression for the output.
4. Simplify the output expression if possible.
5. Implement the circuit for the final, simplified expression.

The following example illustrates the complete design procedure.

EXAMPLE 4-7

Design a logic circuit that has three inputs, A, B, and C, and whose output will be HIGH only when a majority of the inputs are HIGH.

Solution

Step 1. Set up the truth table.

On the basis of the problem statement, the output x should be 1 whenever two or more inputs are 1; for all other cases, the output should be 0 (Table 4-2).

TABLE 4-2

A	B	C	x	
0	0	0	0	
0	0	1	0	
0	1	0	0	
0	1	1	1	$\rightarrow \overline{A}BC$
1	0	0	0	
1	0	1	1	$\rightarrow A\overline{B}C$
1	1	0	1	$\rightarrow AB\overline{C}$
1	1	1	1	$\rightarrow ABC$

Step 2. Write the AND term for each case where the output is a 1.

There are four such cases. The AND terms are shown next to the truth table (Table 4-2). Again note that each AND term contains each input variable in either inverted or noninverted form.

Step 3. Write the sum-of-products expression for the output.

$$x = \overline{A}BC + A\overline{B}C + AB\overline{C} + ABC$$

Step 4. Simplify the output expression.

This expression can be simplified in several ways. Perhaps the quickest way is to realize that the last term ABC has two variables in common with each of the other terms. Thus, we can use the ABC term to factor with each of the other terms. The expression is rewritten with the ABC term occurring three times (recall from Example 4-2 that this is legal in Boolean algebra):

$$x = \overline{A}BC + ABC + A\overline{B}C + ABC + AB\overline{C} + ABC$$

Factoring the appropriate pairs of terms, we have

$$x = BC(\overline{A} + A) + AC(\overline{B} + B) + AB(\overline{C} + C)$$

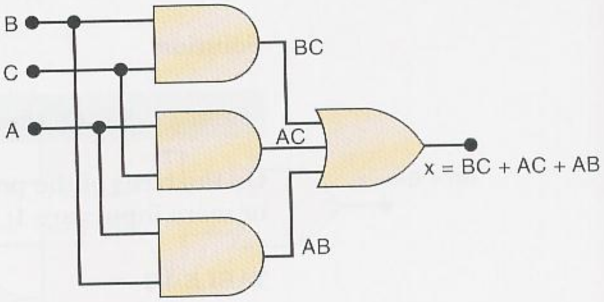
Each term in parentheses is equal to 1, so we have

$$x = BC + AC + AB$$

Step 5. Implement the circuit for the final expression.

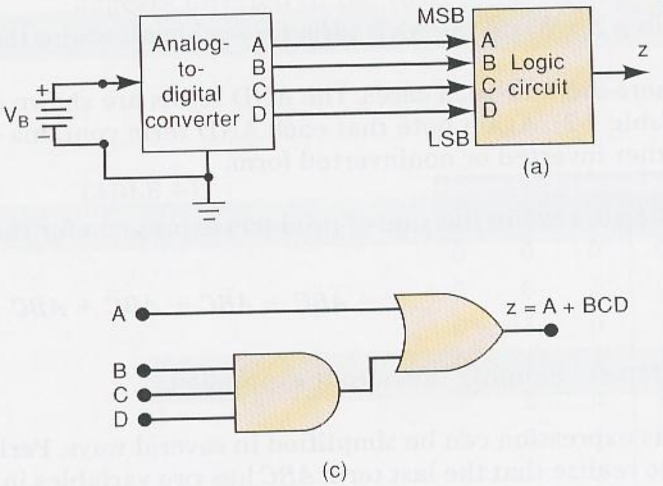
This expression is implemented in Figure 4-7. Since the expression is in SOP form, the circuit consists of a group of AND gates working into a single OR gate.

FIGURE 4-7 Example 4-7.



EXAMPLE 4-8

Refer to Figure 4-8(a), where an analog-to-digital converter is monitoring the dc voltage of a 12-V storage battery on an orbiting spaceship. The converter's output is a four-bit binary number, *ABCD*, corresponding to the battery voltage in steps of 1 V, with *A* as the MSB. The converter's binary outputs are fed to a logic circuit that is to produce a HIGH output as long as the binary value is greater than $0110_2 = 6_{10}$; that is, the battery voltage is greater than 6 V. Design this logic circuit.



	A	B	C	D	z
(0)	0	0	0	0	0
(1)	0	0	0	1	0
(2)	0	0	1	0	0
(3)	0	0	1	1	0
(4)	0	1	0	0	0
(5)	0	1	0	1	0
(6)	0	1	1	0	0
(7)	0	1	1	1	1→ $\overline{A}BCD$
(8)	1	0	0	0	1→ $A\overline{B}\,\overline{C}\,\overline{D}$
(9)	1	0	0	1	1→ $A\overline{B}\,\overline{C}D$
(10)	1	0	1	0	1→ $A\overline{B}C\overline{D}$
(11)	1	0	1	1	1→ $A\overline{B}CD$
(12)	1	1	0	0	1→ $AB\overline{C}\,\overline{D}$
(13)	1	1	0	1	1→ $AB\overline{C}D$
(14)	1	1	1	0	1→ $ABC\overline{D}$
(15)	1	1	1	1	1→ $ABCD$

(b)

FIGURE 4-8 Example 4-8.

Solution

The truth table is shown in Figure 4-8(b). For each case in the truth table, we have indicated the decimal equivalent of the binary number represented by the *ABCD* combination.

The output *z* is set equal to 1 for all those cases where the binary number is greater than 0110. For all other cases, *z* is set equal to 0. This truth table gives us the following sum-of-products expression:

$$\begin{aligned} z = {} & \overline{A}BCD + A\overline{B}\,\overline{C}\,\overline{D} + A\overline{B}\,\overline{C}D + A\overline{B}C\overline{D} + A\overline{B}CD + AB\overline{C}\,\overline{D} \\ & + AB\overline{C}D + ABC\overline{D} + ABCD \end{aligned}$$

Simplification of this expression will be a formidable task, but with a little care it can be accomplished. The step-by-step process involves factoring and eliminating terms of the form $A + \overline{A}$:

$$\begin{aligned}
z &= \overline{A}BCD + A\overline{B}\,\overline{C}(\overline{D} + D) + A\overline{B}C(\overline{D} + D) + AB\overline{C}(\overline{D} + D) + ABC(\overline{D} + D)\\
&= \overline{A}BCD + A\overline{B}\,\overline{C} + A\overline{B}C + AB\overline{C} + ABC\\
&= \overline{A}BCD + A\overline{B}(\overline{C} + C) + AB(\overline{C} + C)\\
&= \overline{A}BCD + A\overline{B} + AB\\
&= \overline{A}BCD + A(\overline{B} + B)\\
&= \overline{A}BCD + A
\end{aligned}$$

This can be reduced further by invoking theorem (15a), which says that $x + \overline{x}y = x + y$. In this case $x = A$ and $y = BCD$. Thus,

$$z = \overline{A}BCD + A = BCD + A$$

This final expression is implemented in Figure 4-8(c).

As this example demonstrates, the algebraic simplification method can be quite lengthy when the original expression contains a large number of terms. This is a limitation that is not shared by the Karnaugh mapping method, as we will see later.

EXAMPLE 4-9

Refer to Figure 4-9(a). In a simple copy machine, a stop signal, S, is to be generated to stop the machine operation and energize an indicator light whenever either of the following conditions exists: (1) there is no paper in the paper feeder tray; or (2) the two microswitches in the paper path are

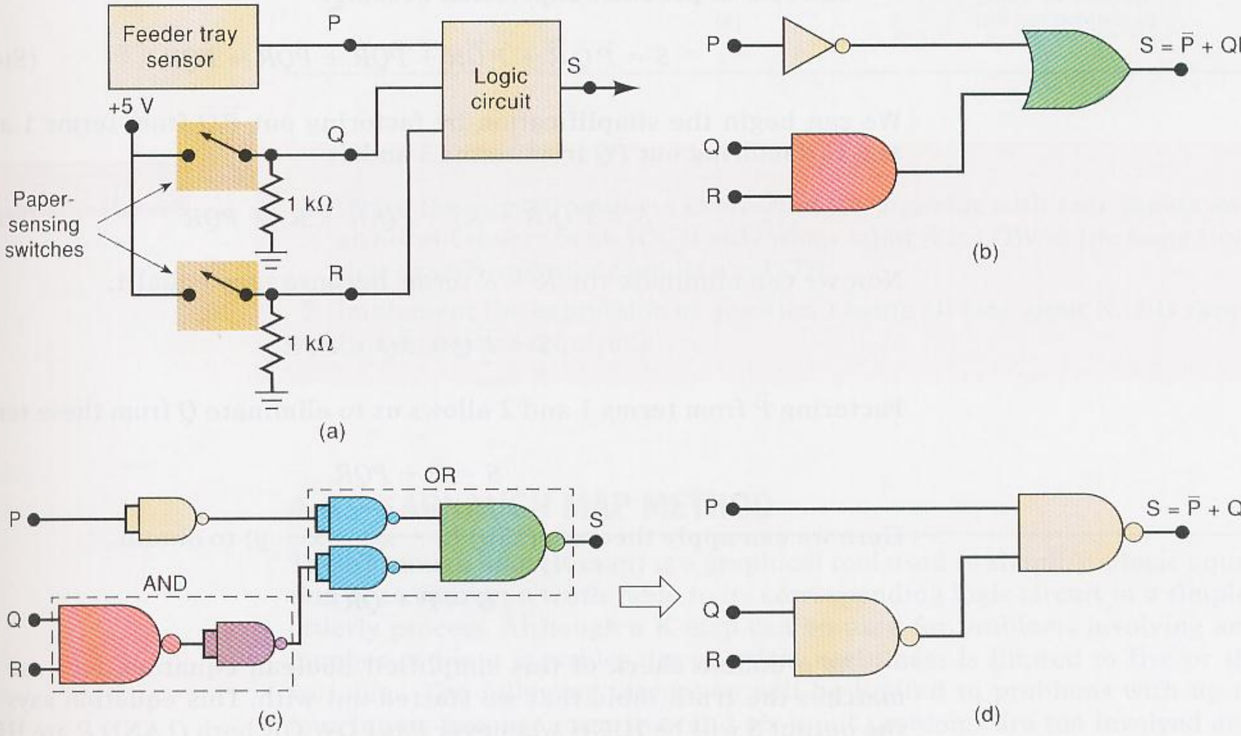


FIGURE 4-9 Example 4-9.

activated, indicating a jam in the paper path. The presence of paper in the feeder tray is indicated by a HIGH at logic signal P. Each of the microswitches produces a logic signal (Q and R) that goes HIGH whenever paper is passing over the switch to activate it. Design the logic circuit to produce a HIGH at output signal S for the stated conditions, and implement it using the 74HC00 CMOS quad two-input NAND chip.

Solution

We will use the five-step process used in Example 4-7. The truth table is shown in Table 4-3. The S output will be a logic 1 whenever $P = 0$ because this indicates no paper in the feeder tray. S will also be a 1 for the two cases where Q and R are both 1, indicating a paper jam. As the table shows, there are five different input conditions that produce a HIGH output. **(Step 1)**

TABLE 4-3

P	Q	R	S	
0	0	0	1	$\overline{P}\,\overline{Q}\,\overline{R}$
0	0	1	1	$\overline{P}\,\overline{Q}R$
0	1	0	1	$\overline{P}Q\overline{R}$
0	1	1	1	$\overline{P}QR$
1	0	0	0	
1	0	1	0	
1	1	0	0	
1	1	1	1	PQR

The AND terms for each of these cases are shown. **(Step 2)**

The sum-of-products expression becomes

$$S = \overline{P}\,\overline{Q}\,\overline{R} + \overline{P}\,\overline{Q}R + \overline{P}Q\overline{R} + \overline{P}QR + PQR$$ **(Step 3)**

We can begin the simplification by factoring out $\overline{P}\,\overline{Q}$ from terms 1 and 2 and by factoring out $\overline{P}Q$ from terms 3 and 4:

$$S = \overline{P}\,\overline{Q}(\overline{R} + R) + \overline{P}Q(\overline{R} + R) + PQR$$ **(Step 4)**

Now we can eliminate the $\overline{R} + R$ terms because they equal 1:

$$S = \overline{P}\,\overline{Q} + \overline{P}Q + PQR$$

Factoring $\overline{P}$ from terms 1 and 2 allows us to eliminate Q from these terms:

$$S = \overline{P} + PQR$$

Here we can apply theorem (15b) ($\overline{x} + xy = \overline{x} + y$) to obtain

$$S = \overline{P} + QR$$

As a double check of this simplified Boolean equation, let's see if it matches the truth table that we started out with. This equation says that the output S will be HIGH whenever P is LOW OR both Q AND R are HIGH. Look at Table 4-3 and observe that the output is HIGH for all four cases when *P* is LOW. *S* is also HIGH when *Q* AND *R* are both HIGH, regardless of the state of *P*. This agrees with the equation.

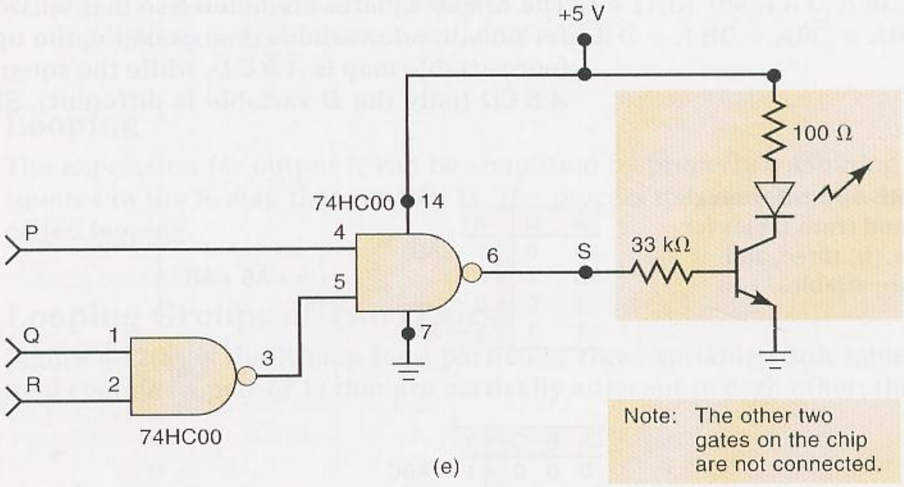
The AND/OR implementation for this circuit is shown in Figure 4-9(b).

(Step 5)

To implement this circuit using the 74HC00 quad two-input NAND chip, we must convert each gate and the INVERTER by their NAND-gate equivalents (per Section 3-12). This is shown in Figure 4-9(c). Clearly, we can eliminate the double inverters to produce the NAND-gate implementation shown in Figure 4-9(d).

The final wired-up circuit is obtained by connecting two of the NAND gates on the 74HC00 chip. This CMOS chip has the same gate configuration and pin numbers as the TTL 74LS00 chip of Figure 3-31. Figure 4-10 shows the wired-up circuit with pin numbers, including the +5 V and GROUND pins. It also includes an output driver transistor and LED to indicate the state of output *S*.

FIGURE 4-10 Circuit of Figure 4-9(d) implemented using 74HC00 NAND chip.



REVIEW QUESTIONS

1. Write the sum-of-products expression for a circuit with four inputs and an output that is to be HIGH only when input *A* is LOW at the same time that exactly two other inputs are LOW.
2. Implement the expression of question 1 using all four-input NAND gates. How many are required?

4-5 KARNAUGH MAP METHOD

The **Karnaugh map (K map)** is a graphical tool used to simplify a logic equation or to convert a truth table to its corresponding logic circuit in a simple, orderly process. Although a K map can be used for problems involving any number of input variables, its practical usefulness is limited to five or six variables. The following discussion will be limited to problems with up to four inputs because even five- and six-input problems are too involved and are best done by a computer program.

Karnaugh Map Format

The K map, like a truth table, is a means for showing the relationship between logic inputs and the desired output. Figure 4-11 shows three examples of K maps for two, three, and four variables, together with the corresponding truth tables. These examples illustrate the following important points:

1. The truth table gives the value of output X for each combination of input values. The K map gives the same information in a different format. Each case in the truth table corresponds to a square in the K map. For example, in Figure 4-11(a), the $A = 0, B = 0$ condition in the truth table corresponds to the $\bar{A}\bar{B}$ square in the K map. Because the truth table shows $X = 1$ for this case, a 1 is placed in the $\bar{A}\bar{B}$ square in the K map. Similarly, the $A = 1, B = 1$ condition in the truth table corresponds to the AB square of the K map. Because $X = 1$ for this case, a 1 is placed in the AB square. All other squares are filled with 0s. This same idea is used in the three- and four-variable maps shown in the figure.
2. The K-map squares are labeled so that horizontally adjacent squares differ only in one variable. For example, the upper left-hand square in the four-variable map is $\bar{A}\,\bar{B}\,\bar{C}\,\bar{D}$, while the square immediately to its right is $\bar{A}\,\bar{B}\,\bar{C}D$ (only the D variable is different). Similarly, vertically adjacent

FIGURE 4-11 Karnaugh maps and truth tables for (a) two, (b) three, and (c) four variables.

A	B	X
0	0	1 → $\bar{A}\bar{B}$
0	1	0
1	0	0
1	1	1 → AB

$$\left\{ X = \bar{A}\bar{B} + AB \right\}$$

	$\bar{B}$	B
$\bar{A}$	1	0
A	0	1

(a)

A	B	C	X
0	0	0	1 → $\bar{A}\bar{B}\bar{C}$
0	0	1	1 → $\bar{A}\bar{B}C$
0	1	0	1 → $\bar{A}B\bar{C}$
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1 → $AB\bar{C}$
1	1	1	0

$$\left\{ \begin{aligned} X &= \bar{A}\bar{B}\bar{C} + \bar{A}\bar{B}C \\ &\quad + \bar{A}B\bar{C} + AB\bar{C} \end{aligned} \right\}$$

	$\bar{C}$	C
$\bar{A}\bar{B}$	1	1
$\bar{A}B$	1	0
AB	1	0
$A\bar{B}$	0	0

(b)

A	B	C	D	X
0	0	0	0	0
0	0	0	1	1 → $\bar{A}\bar{B}\bar{C}D$
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1 → $\bar{A}B\bar{C}D$
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	1 → $AB\bar{C}D$
1	1	1	0	0
1	1	1	1	1 → $ABCD$

$$\left\{ \begin{aligned} X &= \bar{A}\bar{B}\bar{C}D + \bar{A}B\bar{C}D \\ &\quad + AB\bar{C}D + ABCD \end{aligned} \right\}$$

	$\bar{C}\bar{D}$	$\bar{C}D$	CD	$C\bar{D}$
$\bar{A}\bar{B}$	0	1	0	0
$\bar{A}B$	0	1	0	0
AB	0	1	1	0
$A\bar{B}$	0	0	0	0

(c)

squares differ only in one variable. For example, the upper left-hand square is $\overline{A}\,\overline{B}\,\overline{C}\,\overline{D}$, while the square directly below it is $\overline{A}B\overline{C}\,\overline{D}$ (only the B variable is different).

Note that each square in the top row is considered to be adjacent to a corresponding square in the bottom row. For example, the $\overline{A}\,\overline{B}CD$ square in the top row is adjacent to the $A\overline{B}CD$ square in the bottom row because they differ only in the A variable. You can think of the top of the map as being wrapped around to touch the bottom of the map. Similarly, squares in the leftmost column are adjacent to corresponding squares in the rightmost column.

3. In order for vertically and horizontally adjacent squares to differ in only one variable, the top-to-bottom labeling must be done in the order shown: $\overline{A}\,\overline{B}$, $\overline{A}B$, AB, $A\overline{B}$. The same is true of the left-to-right labeling: $\overline{C}\,\overline{D}$, $\overline{C}D$, CD, $C\overline{D}$.
4. Once a K map has been filled with 0s and 1s, the sum-of-products expression for the output X can be obtained by ORing together those squares that contain a 1. In the three-variable map of Figure 4-11(b), the $\overline{A}\,\overline{B}\,\overline{C}$, $\overline{A}\,\overline{B}C$, $\overline{A}B\overline{C}$, and $AB\overline{C}$ squares contain a 1, so that $X = \overline{A}\,\overline{B}\,\overline{C} + \overline{A}\,\overline{B}C + \overline{A}B\overline{C} + AB\overline{C}$.

Looping

The expression for output X can be simplified by properly combining those squares in the K map that contain 1s. The process for combining these 1s is called **looping**.

Looping Groups of Two (Pairs)

Figure 4-12(a) is the K map for a particular three-variable truth table. This map contains a pair of 1s that are vertically adjacent to each other; the first

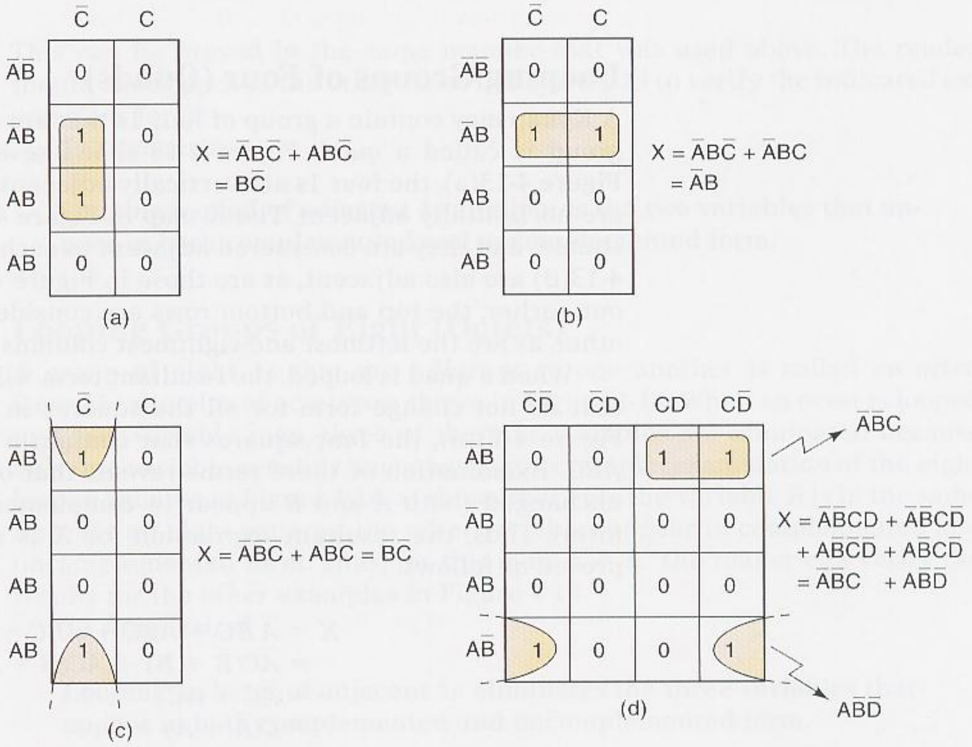

$\overline{C}$
C
$\overline{A}\,\overline{B}$
$\overline{A}B$
AB
$A\overline{B}$
$X = \overline{A}B\overline{C} + AB\overline{C}$
$= B\overline{C}$
(a)
$X = \overline{A}B\overline{C} + \overline{A}BC$
$= \overline{A}B$
(b)
$X = \overline{A}\,\overline{B}\,\overline{C} + A\overline{B}\,\overline{C} = \overline{B}\,\overline{C}$
(c)
$\overline{C}\,\overline{D}$
$\overline{C}D$
CD
$C\overline{D}$
$\overline{A}\,\overline{B}C$
$X = \overline{A}\,\overline{B}CD + \overline{A}\,\overline{B}C\overline{D}$
$+ A\overline{B}\,\overline{C}\,\overline{D} + A\overline{B}C\overline{D}$
$= \overline{A}\,\overline{B}C + A\overline{B}\,\overline{D}$
$A\overline{B}\,\overline{D}$
(d)

FIGURE 4-12 Examples of looping pairs of adjacent 1s.

represents $\overline{A}B\overline{C}$, and the second represents $AB\overline{C}$. Note that in these two terms only the A variable appears in both normal and complemented (inverted) form, while B and $\overline{C}$ remain unchanged. These two terms can be looped (combined) to give a resultant that eliminates the A variable because it appears in both uncomplemented and complemented forms. This is easily proved as follows:

$$\begin{aligned} X &= \overline{A}B\overline{C} + AB\overline{C} \\ &= B\overline{C}(\overline{A} + A) \\ &= B\overline{C}(1) = B\overline{C} \end{aligned}$$

This same principle holds true for any pair of vertically or horizontally adjacent 1s. Figure 4-12(b) shows an example of two horizontally adjacent 1s. These two can be looped and the C variable eliminated because it appears in both its uncomplemented and complemented forms to give a resultant of $X = \overline{A}B$.

Another example is shown in Figure 4-12(c). In a K map, the top row and bottom row of squares are considered to be adjacent. Thus, the two 1s in this map can be looped to provide a resultant of $\overline{A}\,\overline{B}\,\overline{C} + A\overline{B}\,\overline{C} = \overline{B}\,\overline{C}$.

Figure 4-12(d) shows a K map that has two pairs of 1s that can be looped. The two 1s in the top row are horizontally adjacent. The two 1s in the bottom row are also adjacent because, in a K map, the leftmost column and the rightmost column of squares are considered to be adjacent. When the top pair of 1s is looped, the D variable is eliminated (because it appears as both D and $\overline{D}$) to give the term $\overline{A}\,\overline{B}C$. Looping the bottom pair eliminates the C variable to give the term $A\overline{B}\,\overline{D}$. These two terms are ORed to give the final result for X.

To summarize:

Looping a pair of adjacent 1s in a K map eliminates the variable that appears in complemented and uncomplemented form.

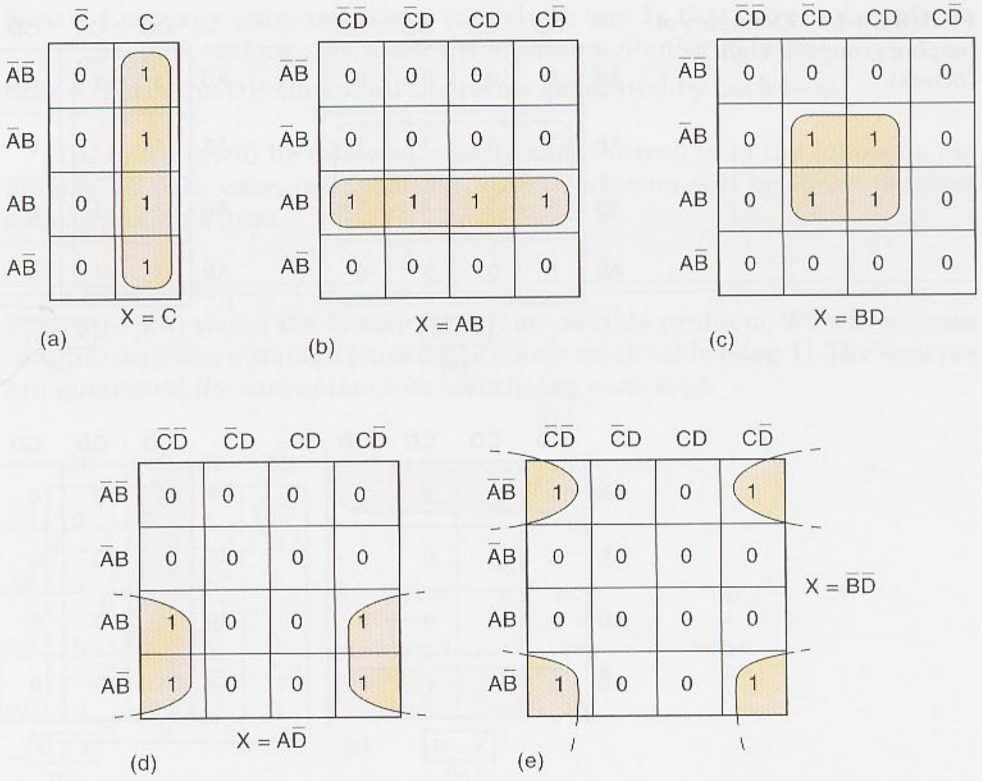
Looping Groups of Four (Quads)

A K map may contain a group of four 1s that are adjacent to each other. This group is called a *quad.* Figure 4-13 shows several examples of quads. In Figure 4-13(a), the four 1s are vertically adjacent, and in Figure 4-13(b), they are horizontally adjacent. The K map in Figure 4-13(c) contains four 1s in a square, and they are considered adjacent to each other. The four 1s in Figure 4-13(d) are also adjacent, as are those in Figure 4-13(e), because, as pointed out earlier, the top and bottom rows are considered to be adjacent to each other, as are the leftmost and rightmost columns.

When a quad is looped, the resultant term will contain only the variables that do not change form for all the squares in the quad. For example, in Figure 4-13(a), the four squares that contain a 1 are $\overline{A}\,\overline{B}C$, $\overline{A}BC$, ABC, and $A\overline{B}C$. Examination of these terms reveals that only the variable C remains unchanged (both A and B appear in complemented and uncomplemented form). Thus, the resultant expression for X is simply $X = C$. This can be proved as follows:

$$\begin{aligned} X &= \overline{A}\,\overline{B}C + \overline{A}BC + ABC + A\overline{B}C \\ &= \overline{A}C(\overline{B} + B) + AC(B + \overline{B}) \\ &= \overline{A}C + AC \\ &= C(\overline{A} + A) = C \end{aligned}$$

FIGURE 4-13 Examples of looping groups of four 1s (quads).


$\bar{C}$ C
$\bar{A}\bar{B}$ 0 1
$\bar{A}B$ 0 1
AB 0 1
$A\bar{B}$ 0 1
X = C
(a)
$\bar{C}\bar{D}$ $\bar{C}D$ CD $C\bar{D}$
$\bar{A}\bar{B}$ 0 0 0 0
$\bar{A}B$ 0 0 0 0
AB 1 1 1 1
$A\bar{B}$ 0 0 0 0
X = AB
(b)
$\bar{C}\bar{D}$ $\bar{C}D$ CD $C\bar{D}$
$\bar{A}\bar{B}$ 0 0 0 0
$\bar{A}B$ 0 1 1 0
AB 0 1 1 0
$A\bar{B}$ 0 0 0 0
X = BD
(c)
$\bar{C}\bar{D}$ $\bar{C}D$ CD $C\bar{D}$
$\bar{A}\bar{B}$ 0 0 0 0
$\bar{A}B$ 0 0 0 0
AB 1 0 0 1
$A\bar{B}$ 1 0 0 1
X = $A\bar{D}$
(d)
$\bar{C}\bar{D}$ $\bar{C}D$ CD $C\bar{D}$
$\bar{A}\bar{B}$ 1 0 0 1
$\bar{A}B$ 0 0 0 0
AB 0 0 0 0
$A\bar{B}$ 1 0 0 1
X = $\bar{B}\bar{D}$
(e)

As another example, consider Figure 4-13(d), where the four squares containing 1s are $AB\bar{C}\bar{D}$, $A\bar{B}\bar{C}\bar{D}$, $ABC\bar{D}$, and $A\bar{B}C\bar{D}$. Examination of these terms indicates that only the variables A and $\bar{D}$ remain unchanged, so that the simplified expression for X is

$$X = A\bar{D}$$

This can be proved in the same manner that was used above. The reader should check each of the other cases in Figure 4-13 to verify the indicated expressions for X.

To summarize:

Looping a quad of adjacent 1s eliminates the two variables that appear in both complemented and uncomplemented form.

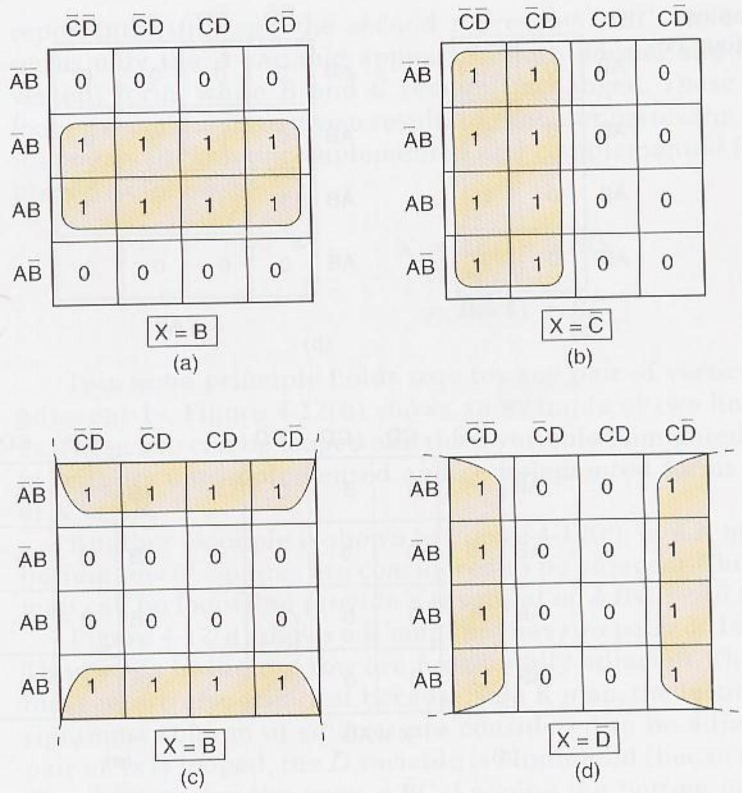
Looping Groups of Eight (Octets)

A group of eight 1s that are adjacent to one another is called an *octet*. Several examples of octets are shown in Figure 4-14. When an octet is looped in a four-variable map, three of the four variables are eliminated because only one variable remains unchanged. For example, examination of the eight looped squares in Figure 4-14(a) shows that only the variable B is in the same form for all eight squares: the other variables appear in complemented and uncomplemented form. Thus, for this map, $X = B$. The reader can verify the results for the other examples in Figure 4-14.

To summarize:

Looping an octet of adjacent 1s eliminates the three variables that appear in both complemented and uncomplemented form.

FIGURE 4-14 Examples of looping groups of eight 1s (octets).



	$\overline{C}\overline{D}$	$\overline{C}D$	CD	$C\overline{D}$
$\overline{A}\overline{B}$	0	0	0	0
$\overline{A}B$	1	1	1	1
AB	1	1	1	1
$A\overline{B}$	0	0	0	0

$X = B$

(a)

	$\overline{C}\overline{D}$	$\overline{C}D$	CD	$C\overline{D}$
$\overline{A}\overline{B}$	1	1	0	0
$\overline{A}B$	1	1	0	0
AB	1	1	0	0
$A\overline{B}$	1	1	0	0

$X = \overline{C}$

(b)

	$\overline{C}\overline{D}$	$\overline{C}D$	CD	$C\overline{D}$
$\overline{A}\overline{B}$	1	1	1	1
$\overline{A}B$	0	0	0	0
AB	0	0	0	0
$A\overline{B}$	1	1	1	1

$X = \overline{B}$

(c)

	$\overline{C}\overline{D}$	$\overline{C}D$	CD	$C\overline{D}$
$\overline{A}\overline{B}$	1	0	0	1
$\overline{A}B$	1	0	0	1
AB	1	0	0	1
$A\overline{B}$	1	0	0	1

$X = \overline{D}$

(d)

Complete Simplification Process

We have seen how looping of pairs, quads, and octets on a K map can be used to obtain a simplified expression. We can summarize the rule for loops of *any* size:

> **When a variable appears in both complemented and uncomplemented form within a loop, that variable is eliminated from the expression. Variables that are the same for all squares of the loop must appear in the final expression.**

It should be clear that a larger loop of 1s eliminates more variables. To be exact, a loop of two eliminates one variable, a loop of four eliminates two variables, and a loop of eight eliminates three. This principle will now be used to obtain a simplified logic expression from a K map that contains any combination of 1s and 0s.

The procedure will first be outlined and then applied to several examples. The steps below are followed in using the K-map method for simplifying a Boolean expression:

Step 1 Construct the K map and place 1s in those squares corresponding to the 1s in the truth table. Place 0s in the other squares.

Step 2 Examine the map for adjacent 1s and loop those 1s that are *not* adjacent to any other 1s. These are called *isolated* 1s.

Step 3 Next, look for those 1s that are adjacent to only one other 1. Loop *any* pair containing such a 1.

Step 4 Loop any octet even if it contains some 1s that have already been looped.

Step 5 Loop any quad that contains one or more 1s that have not already been looped, *making sure to use the minimum number of loops.*

Step 6 Loop any pairs necessary to include any 1s that have not yet been looped, *making sure to use the minimum number of loops.*

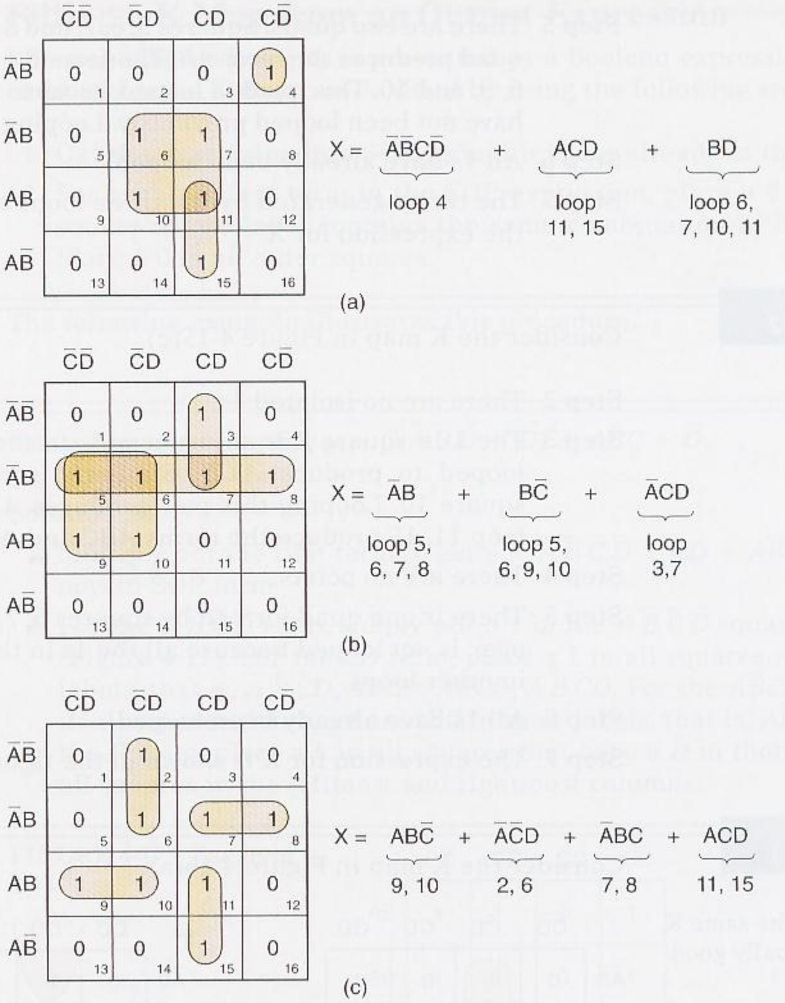
Step 7 Form the OR sum of all the terms generated by each loop.

These steps will be followed exactly and referred to in the following examples. In each case, the resulting logic expression will be in its simplest sum-of-products form.

EXAMPLE 4-10

Figure 4-15(a) shows the K map for a four-variable problem. We will assume that the map was obtained from the problem truth table (step 1). The squares are numbered for convenience in identifying each loop.

FIGURE 4-15 Examples 4-10 to 4-12.



	$\overline{C}\overline{D}$	$\overline{C}D$	CD	$C\overline{D}$
$\overline{A}\overline{B}$	0	0	0	1
$\overline{A}B$	0	1	1	0
AB	0	1	1	0
$A\overline{B}$	0	0	1	0

$$X = \underbrace{\overline{A}\,\overline{B}C\overline{D}}_{\text{loop 4}} + \underbrace{ACD}_{\text{loop 11, 15}} + \underbrace{BD}_{\text{loop 6, 7, 10, 11}}$$

(a)

	$\overline{C}\overline{D}$	$\overline{C}D$	CD	$C\overline{D}$
$\overline{A}\overline{B}$	0	0	1	0
$\overline{A}B$	1	1	1	1
AB	1	1	0	0
$A\overline{B}$	0	0	0	0

$$X = \underbrace{\overline{A}B}_{\text{loop 5, 6, 7, 8}} + \underbrace{B\overline{C}}_{\text{loop 5, 6, 9, 10}} + \underbrace{\overline{A}CD}_{\text{loop 3,7}}$$

(b)

	$\overline{C}\overline{D}$	$\overline{C}D$	CD	$C\overline{D}$
$\overline{A}\overline{B}$	0	1	0	0
$\overline{A}B$	0	1	1	1
AB	1	1	1	0
$A\overline{B}$	0	0	1	0

$$X = \underbrace{AB\overline{C}}_{9,\,10} + \underbrace{\overline{A}\,\overline{C}D}_{2,\,6} + \underbrace{\overline{A}BC}_{7,\,8} + \underbrace{ACD}_{11,\,15}$$

(c)

Step 2 Square 4 is the only square containing a 1 that is not adjacent to any other 1. It is looped and is referred to as loop 4.

Step 3 Square 15 is adjacent *only* to square 11. This pair is looped and referred to as loop 11, 15.

Step 4 There are no octets.

Step 5 Squares 6, 7, 10, and 11 form a quad. This quad is looped (loop 6, 7, 10, 11). Note that square 11 is used again, even though it was part of loop 11, 15.

Step 6 All 1s have already been looped.

Step 7 Each loop generates a term in the expression for X. Loop 4 is simply $\overline{A}\,\overline{B}C\overline{D}$. Loop 11, 15 is ACD (the B variable is eliminated). Loop 6, 7, 10, 11 is BD (A and C are eliminated).

EXAMPLE 4-11

Consider the K map in Figure 4-15(b). Once again, we can assume that step 1 has already been performed.

Step 2 There are no isolated 1s.

Step 3 The 1 in square 3 is adjacent *only* to the 1 in square 7. Looping this pair (loop 3, 7) produces the term $\overline{A}CD$.

Step 4 There are no octets.

Step 5 There are two quads. Squares 5, 6, 7, and 8 form one quad. Looping this quad produces the term $\overline{A}B$. The second quad is made up of squares 5, 6, 9, and 10. This quad is looped because it contains two squares that have not been looped previously. Looping this quad produces $B\overline{C}$.

Step 6 All 1s have already been looped.

Step 7 The terms generated by the three loops are ORed together to obtain the expression for X.

EXAMPLE 4-12

Consider the K map in Figure 4-15(c).

Step 2 There are no isolated 1s.

Step 3 The 1 in square 2 is adjacent only to the 1 in square 6. This pair is looped to produce $\overline{A}\,\overline{C}D$. Similarly, square 9 is adjacent only to square 10. Looping this pair produces $AB\overline{C}$. Likewise, loop 7, 8 and loop 11, 15 produce the terms $\overline{A}BC$ and ACD, respectively.

Step 4 There are no octets.

Step 5 There is one quad formed by squares 6, 7, 10, and 11. This quad, however, is *not* looped because all the 1s in the quad have been included in other loops.

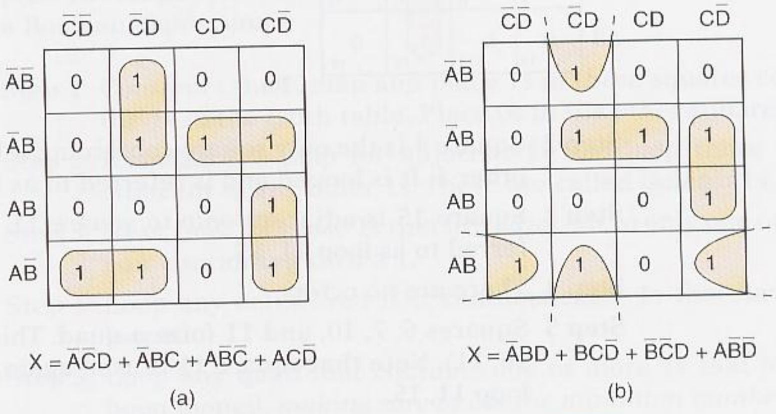
Step 6 All 1s have already been looped.

Step 7 The expression for X is shown in the figure.

EXAMPLE 4-13

Consider the K map in Figure 4-16(a).

FIGURE 4-16 The same K map with two equally good solutions.



$X = \overline{A}\,\overline{C}D + \overline{A}BC + A\overline{B}\,\overline{C} + AC\overline{D}$

(a)

$X = \overline{A}BD + BC\overline{D} + \overline{B}\,\overline{C}D + A\overline{B}\,\overline{D}$

(b)

Step 2 There are no isolated 1s.

Step 3 There are no 1s that are adjacent to only one other 1.

Step 4 There are no octets.

Step 5 There are no quads.

Steps 6 and 7 There are many possible pairs. The looping must use the minimum number of loops to account for all the 1s. For this map, there are *two* possible loopings, which require only four looped pairs. Figure 4-16(a) shows one solution and its resultant expression. Figure 4-16(b) shows the other. Note that both expressions are of the same complexity, and so neither is better than the other.

Filling a K Map from an Output Expression

When the desired output is presented as a Boolean expression instead of a truth table, the K map can be filled by using the following steps:

1. Get the expression into SOP form if it is not already in that form.
2. For each product term in the SOP expression, place a 1 in each K-map square whose label contains the same combination of input variables. Place a 0 in all other squares.

The following example illustrates this procedure.

EXAMPLE 4-14

Use a K map to simplify $y = \overline{C}(\overline{A}\,\overline{B}\,\overline{D} + D) + A\overline{B}C + \overline{D}$.

Solution

1. Multiply out the first term to get $y = \overline{A}\,\overline{B}\,\overline{C}\,\overline{D} + \overline{C}D + A\overline{B}C + \overline{D}$, which is now in SOP form.
2. For the $\overline{A}\,\overline{B}\,\overline{C}\,\overline{D}$ term, simply put a 1 in the $\overline{A}\,\overline{B}\,\overline{C}\,\overline{D}$ square of the K map (Figure 4-17). For the $\overline{C}D$ term, place a 1 in all squares with $\overline{C}D$ in their labels, that is, $\overline{A}\,\overline{B}\,\overline{C}D$, $\overline{A}B\overline{C}D$, $AB\overline{C}D$, $A\overline{B}\,\overline{C}D$. For the $A\overline{B}C$ term, place a 1 in all squares that have an $A\overline{B}C$ in their labels, that is, $A\overline{B}C\overline{D}$, $A\overline{B}CD$. For the $\overline{D}$ term, place a 1 in all squares that have a $\overline{D}$ in their labels, that is, all squares in the leftmost and rightmost columns.

FIGURE 4-17 Example 4-14.

	$\overline{C}\,\overline{D}$	$\overline{C}D$	CD	$C\overline{D}$
$\overline{A}\,\overline{B}$	1	1	0	1
$\overline{A}B$	1	1	0	1
AB	1	1	0	1
$A\overline{B}$	1	1	1	1

$y = A\overline{B} + \overline{C} + \overline{D}$

The K map is now filled and can be looped for simplification. Verify that proper looping produces $y = A\overline{B} + \overline{C} + \overline{D}$.

Don't-Care Conditions

Some logic circuits can be designed so that there are certain input conditions for which there are no specified output levels, usually because these input conditions will never occur. In other words, there will be certain combinations of input levels where we "don't care" whether the output is HIGH or LOW. This is illustrated in the truth table of Figure 4-18(a).

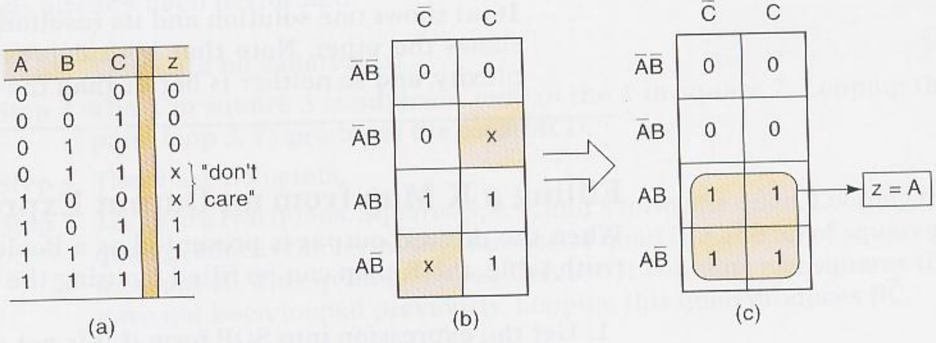


FIGURE 4-18 "Don't-care" conditions should be changed to 0 or 1 to produce K-map looping that yields the simplest expression.

Here the output z is not specified as either 0 or 1 for the conditions $A, B, C = 1, 0, 0$ and $A, B, C = 0, 1, 1$. Instead, an x is shown for these conditions. The x represents the **don't-care condition**. A don't-care condition can come about for several reasons, the most common being that in some situations certain input combinations can never occur, and so there is no specified output for these conditions.

A circuit designer is free to make the output for any don't-care condition either a 0 or a 1 to produce the simplest output expression. For example, the K map for this truth table is shown in Figure 4-18(b) with an x placed in the $A\overline{B}\,\overline{C}$ and $\overline{A}BC$ squares. The designer here would be wise to change the x in the $A\overline{B}\,\overline{C}$ square to a 1 and the x in the $\overline{A}BC$ square to a 0 because this would produce a quad that can be looped to produce $z = A$, as shown in Figure 4-18(c).

Whenever don't-care conditions occur, we must decide which x to change to 0 and which to 1 to produce the best K-map looping (i.e., the simplest expression). This decision is not always an easy one. Several end-of-chapter problems will provide practice in dealing with don't-care cases. Here's another example.

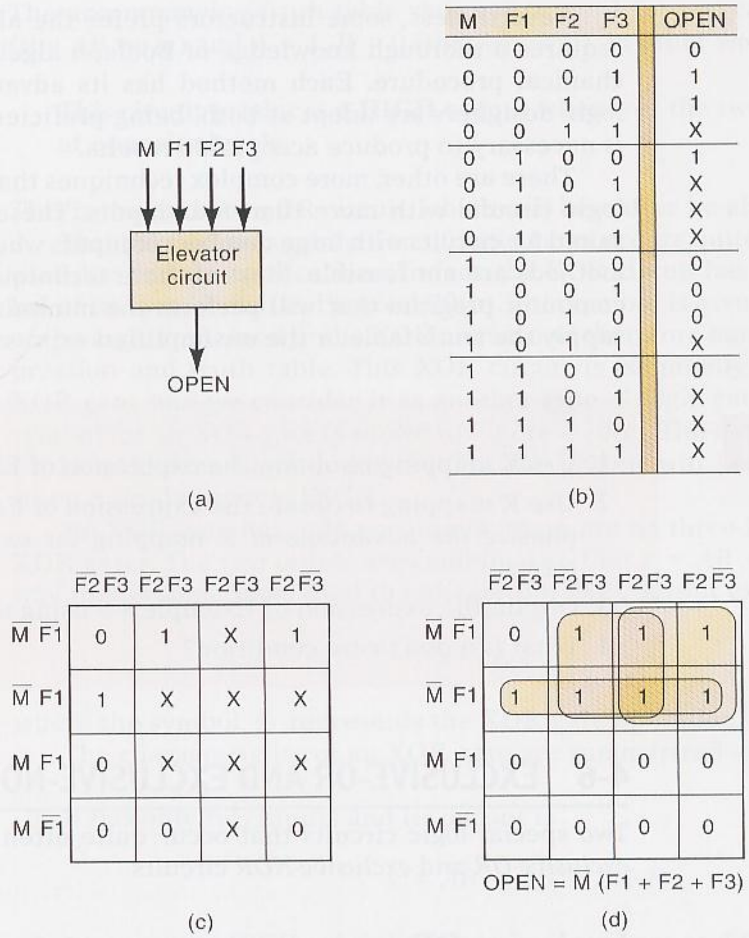
EXAMPLE 4-15

Let's design a logic circuit that controls an elevator door in a three-story building. The circuit in Figure 4-19(a) has four inputs. M is a logic signal that indicates when the elevator is moving ($M = 1$) or stopped ($M = 0$). $F1$, $F2$, and $F3$ are floor indicator signals that are normally LOW, and they go HIGH only when the elevator is positioned at the level of that particular floor. For example, when the elevator is lined up level with the second floor, $F2 = 1$ and $F1 = F3 = 0$. The circuit output is the *OPEN* signal, which is normally LOW and will go HIGH when the elevator door is to be opened.

We can fill in the truth table for the *OPEN* output [Figure 4-19(b)] as follows:

1. Because the elevator cannot be lined up with more than one floor at a time, only one of the floor inputs can be HIGH at any given time. This means that all those cases in the truth table where more than one floor

FIGURE 4-19 Example 4-15.



(a)

M	F1	F2	F3	OPEN
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	X
0	1	0	0	1
0	1	0	1	X
0	1	1	0	X
0	1	1	1	X
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	X
1	1	0	0	0
1	1	0	1	X
1	1	1	0	X
1	1	1	1	X

(b)

	$\overline{F2}\,\overline{F3}$	$\overline{F2}F3$	$F2F3$	$F2\overline{F3}$
$\overline{M}\,\overline{F1}$	0	1	X	1
$\overline{M}F1$	1	X	X	X
$MF1$	0	X	X	X
$M\overline{F1}$	0	0	X	0

(c)

	$\overline{F2}\,\overline{F3}$	$\overline{F2}F3$	$F2F3$	$F2\overline{F3}$
$\overline{M}\,\overline{F1}$	0	1	1	1
$\overline{M}F1$	1	1	1	1
$MF1$	0	0	0	0
$M\overline{F1}$	0	0	0	0

$OPEN = \overline{M}\,(F1 + F2 + F3)$

(d)

input is a 1 are don't-care conditions. We can place an *x* in the *OPEN* output column for those eight cases where more than one *F* input is 1.

2. Looking at the other eight cases, when $M = 1$ the elevator is moving, so *OPEN* must be a 0 because we do not want the elevator door to open. When $M = 0$ (elevator stopped) we want $OPEN = 1$ provided that one of the floor inputs is 1. When $M = 0$ and all floor inputs are 0, the elevator is stopped but is not properly lined up with any floor, so we want $OPEN = 0$ to keep the door closed.

The truth table is now complete and we can transfer its information to the K map in Figure 4-19(c). The map has only three 1s, but it has eight don't-cares. By changing four of these don't-care squares to 1s, we can produce quad loopings that contain the original 1s [Figure 4-19(d)]. This is the best we can do as far as minimizing the output expression. Verify that the loopings produce the *OPEN* output expression shown.

Summary

The K-map process has several advantages over the algebraic method. K mapping is a more orderly process with well-defined steps compared with the trial-and-error process sometimes used in algebraic simplification. K mapping usually requires fewer steps, especially for expressions containing many terms, and it always produces a minimum expression.

Nevertheless, some instructors prefer the algebraic method because it requires a thorough knowledge of Boolean algebra and is not simply a mechanical procedure. Each method has its advantages, and although most logic designers are adept at both, being proficient in one method is all that is necessary to produce acceptable results.

There are other, more complex techniques that designers use to minimize logic circuits with more than four inputs. These techniques are especially suited for circuits with large numbers of inputs where algebraic and K-mapping methods are not feasible. Most of these techniques can be translated into a computer program that will perform the minimization from input data that supply the truth table or the unsimplified expression.

REVIEW QUESTIONS

1. Use K mapping to obtain the expression of Example 4-7.
2. Use K mapping to obtain the expression of Example 4-8. This should emphasize the advantage of K mapping for expressions containing many terms.
3. Obtain the expression of Example 4-9 using a K map.
4. What is a don't-care condition?

4-6 EXCLUSIVE-OR AND EXCLUSIVE-NOR CIRCUITS

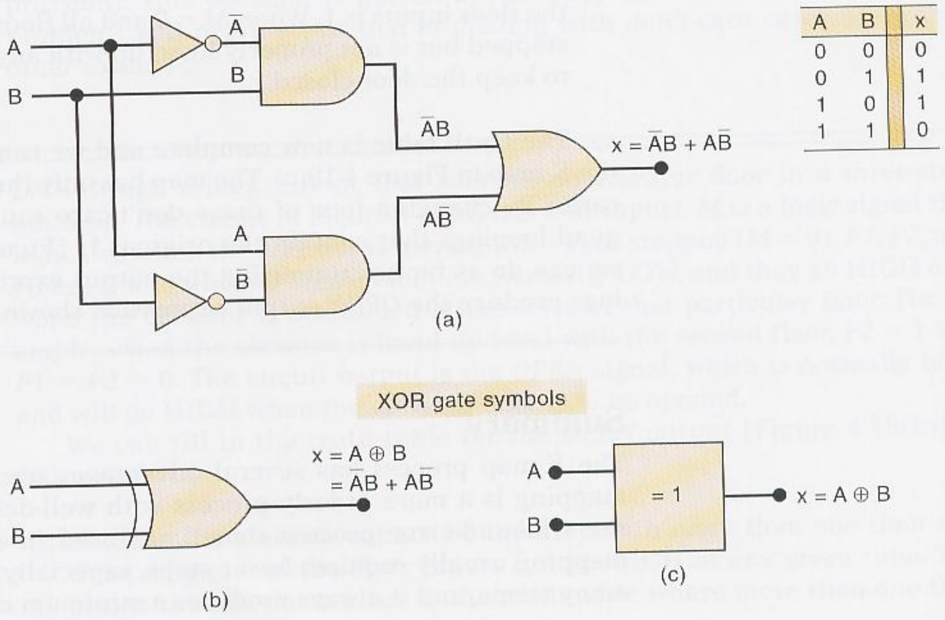
Two special logic circuits that occur quite often in digital systems are the *exclusive-OR* and *exclusive-NOR* circuits.

Exclusive-OR

Consider the logic circuit of Figure 4-20(a). The output expression of this circuit is

$$x = \overline{A}B + A\overline{B}$$

FIGURE 4-20
(a) Exclusive-OR circuit and truth table; (b) traditional XOR gate symbol; (c) IEEE/ANSI symbol for XOR gate.



A	B	x
0	0	0
0	1	1
1	0	1
1	1	0

The accompanying truth table shows that $x = 1$ for two cases: $A = 0, B = 1$ (the $\overline{A}B$ term) and $A = 1, B = 0$ (the $A\overline{B}$ term). In other words:

> **This circuit produces a HIGH output whenever the two inputs are at opposite levels.**

This is the **exclusive-OR** circuit, which will hereafter be abbreviated **XOR**.

This particular combination of logic gates occurs quite often and is very useful in certain applications. In fact, the XOR circuit has been given a symbol of its own, shown in Figure 4-20(b). This symbol is assumed to contain all of the logic contained in the XOR circuit and therefore has the same logic expression and truth table. This XOR circuit is commonly referred to as an XOR *gate,* and we consider it as another type of logic gate. The IEEE/ANSI symbol for an XOR gate is shown in Figure 4-20(c). The dependency notation (= 1) inside the block indicates that the output will be active-HIGH *only* when a single input is HIGH.

An XOR gate has only *two* inputs; there are no three-input or four-input XOR gates. The two inputs are combined so that $x = \overline{A}B + A\overline{B}$. A shorthand way that is sometimes used to indicate the XOR output expression is

$$x = A \oplus B$$

where the symbol $\oplus$ represents the XOR gate operation.

The characteristics of an XOR gate are summarized as follows:

1. It has only two inputs and its output is

$$x = \overline{A}B + A\overline{B} = A \oplus B$$

2. Its output is *HIGH* only when the two inputs are at *different* levels.

Several ICs are available that contain XOR gates. Those listed below are *quad* XOR chips containing four XOR gates.

74LS86	Quad XOR (TTL family)
74C86	Quad XOR (CMOS family)
74HC86	Quad XOR (high-speed CMOS)

Exclusive-NOR

The **exclusive-NOR** circuit (abbreviated **XNOR**) operates completely opposite to the XOR circuit. Figure 4-21(a) shows an XNOR circuit and its accompanying truth table. The output expression is

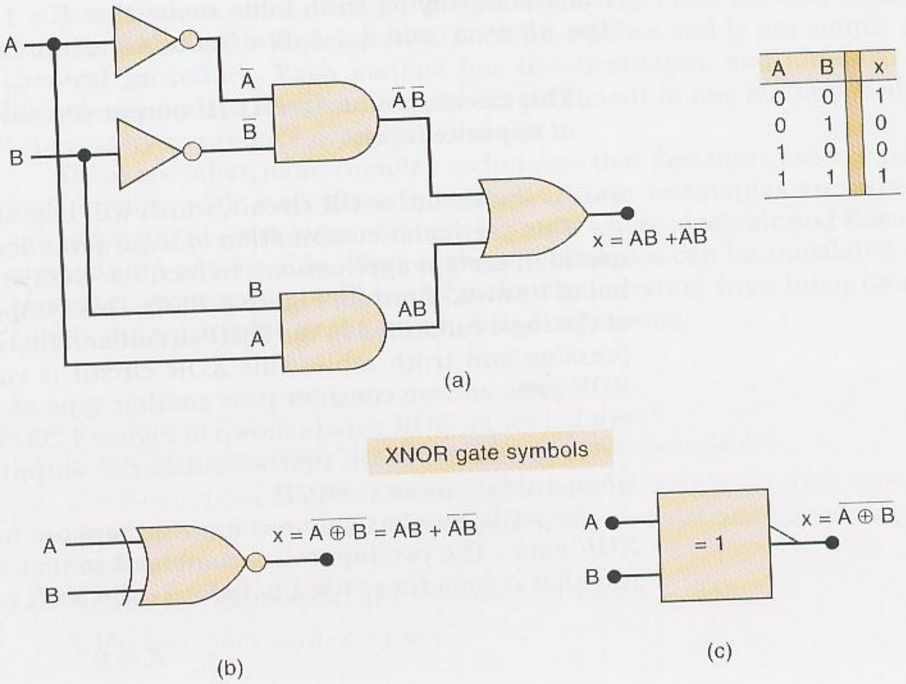
$$x = AB + \overline{A}\,\overline{B}$$

which indicates along with the truth table that x will be 1 for two cases: $A = B = 1$ (the AB term) and $A = B = 0$ (the $\overline{A}\,\overline{B}$ term). In other words:

> **The XNOR produces a HIGH output whenever the two inputs are at the same level.**

It should be apparent that the output of the XNOR circuit is the exact inverse of the output of the XOR circuit. The traditional symbol for an XNOR

FIGURE 4-21 (a) Exclusive-NOR circuit; (b) traditional symbol for XNOR gate; (c) IEEE/ANSI symbol.


A
B
$\overline{A}$
$\overline{B}$
$\overline{A}\,\overline{B}$
$x = AB + \overline{A}\,\overline{B}$
B
AB
A
(a)
XNOR gate symbols
A
B
$x = \overline{A \oplus B} = AB + \overline{A}\,\overline{B}$
(b)
A
B
= 1
$x = \overline{A \oplus B}$
(c)

A	B	x
0	0	1
0	1	0
1	0	0
1	1	1

gate is obtained by simply adding a small circle at the output of the XOR symbol [Figure 4-21(b)]. The IEEE/ANSI symbol adds the small triangle on the output of the XOR symbol. Both symbols indicate an output that goes to its active-LOW state when *only one* input is HIGH.

The XNOR gate also has *only two* inputs, and it combines them so that its output is

$$x = AB + \overline{A}\,\overline{B}$$

A shorthand way to indicate the output expression of the XNOR is

$$x = \overline{A \oplus B}$$

which is simply the inverse of the XOR operation. The XNOR gate is summarized as follows:

1. It has only two inputs and its output is

$$x = AB + \overline{A}\,\overline{B} = \overline{A \oplus B}$$

2. Its output is HIGH only when the two inputs are at the *same* level.

Several ICs are available that contain XNOR gates. Those listed below are quad XNOR chips containing four XNOR gates.

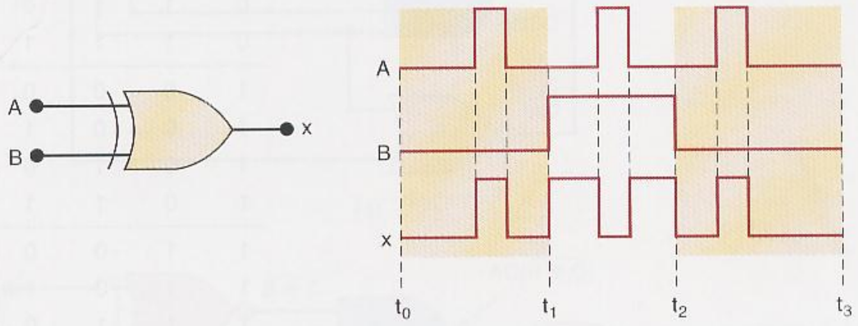
74LS266	Quad XNOR (TTL family)
74C266	Quad XNOR (CMOS)
74HC266	Quad XNOR (high-speed CMOS)

Each of these XNOR chips, however, has special output circuitry that limits its use to special types of applications. Very often, a logic designer will obtain the XNOR function simply by connecting the output of an XOR to an INVERTER.

EXAMPLE 4-16

Determine the output waveform for the input waveforms given in Figure 4-22.

FIGURE 4-22 Example 4-16.



Solution

The output waveform is obtained using the fact that the XOR output will go HIGH only when its inputs are at different levels. The resulting output waveform reveals several interesting points:

1. The x waveform matches the A input waveform during those time intervals when $B = 0$. This occurs during the time intervals t_0 to t_1 and t_2 to t_3.
2. The x waveform is the *inverse* of the A input waveform during those time intervals when $B = 1$. This occurs during the interval t_1 to t_2.
3. These observations show that an XOR gate can be used as a *controlled INVERTER*; that is, one of its inputs can be used to control whether or not the signal at the other input will be inverted. This property will be useful in certain applications.

EXAMPLE 4-17

The notation x_1x_0 represents a two-bit binary number that can have any value (00, 01, 10, or 11); for example, when $x_1 = 1$ and $x_0 = 0$, the binary number is 10, and so on. Similarly, y_1y_0 represents another two-bit binary number. Design a logic circuit, using x_1, x_0, y_1, and y_0 inputs, whose output will be HIGH only when the two binary numbers x_1x_0 and y_1y_0 are *equal*.

Solution

The first step is to construct a truth table for the 16 input conditions (Table 4-4). The output z must be HIGH whenever the x_1x_0 values match the y_1y_0 values; that is, whenever $x_1 = y_1$ and $x_0 = y_0$. The table shows that there are four such cases. We could now continue with the normal procedure, which would be to obtain a sum-of-products expression for z, attempt to simplify it, and then implement the result. However, the nature of this problem makes it ideally suited for implementation using XNOR gates, and a little thought

TABLE 4-4

x_1	x_0	y_1	y_0	z (Output)
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

will produce a simple solution with minimum work. Refer to Figure 4-23; in this logic diagram, x_1 and y_1 are fed to one XNOR gate, and x_0 and y_0 are fed to another XNOR gate. The output of each XNOR will be HIGH only when its inputs are equal. Thus, for $x_0 = y_0$ and $x_1 = y_1$, both XNOR outputs will be HIGH. This is the condition we are looking for because it means that the two two-bit numbers are equal. The AND gate output will be HIGH only for this case, thereby producing the desired output.

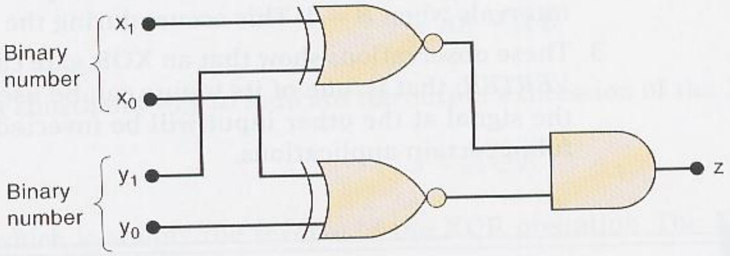


FIGURE 4-23 Circuit for detecting equality of two two-bit binary numbers.

EXAMPLE 4-18

When simplifying the expression for the output of a combinational logic circuit, you may encounter the XOR or XNOR operations as you are factoring. This will often lead to the use of XOR or XNOR gates in the implementation of the final circuit. To illustrate, simplify the circuit of Figure 4-24(a).

Solution

The unsimplified expression for the circuit is obtained as

$$z = ABCD + A\overline{B}\,\overline{C}D + \overline{A}\,\overline{D}$$

We can factor AD from the first two terms:

$$z = AD(BC + \overline{B}\,\overline{C}) + \overline{A}\,\overline{D}$$

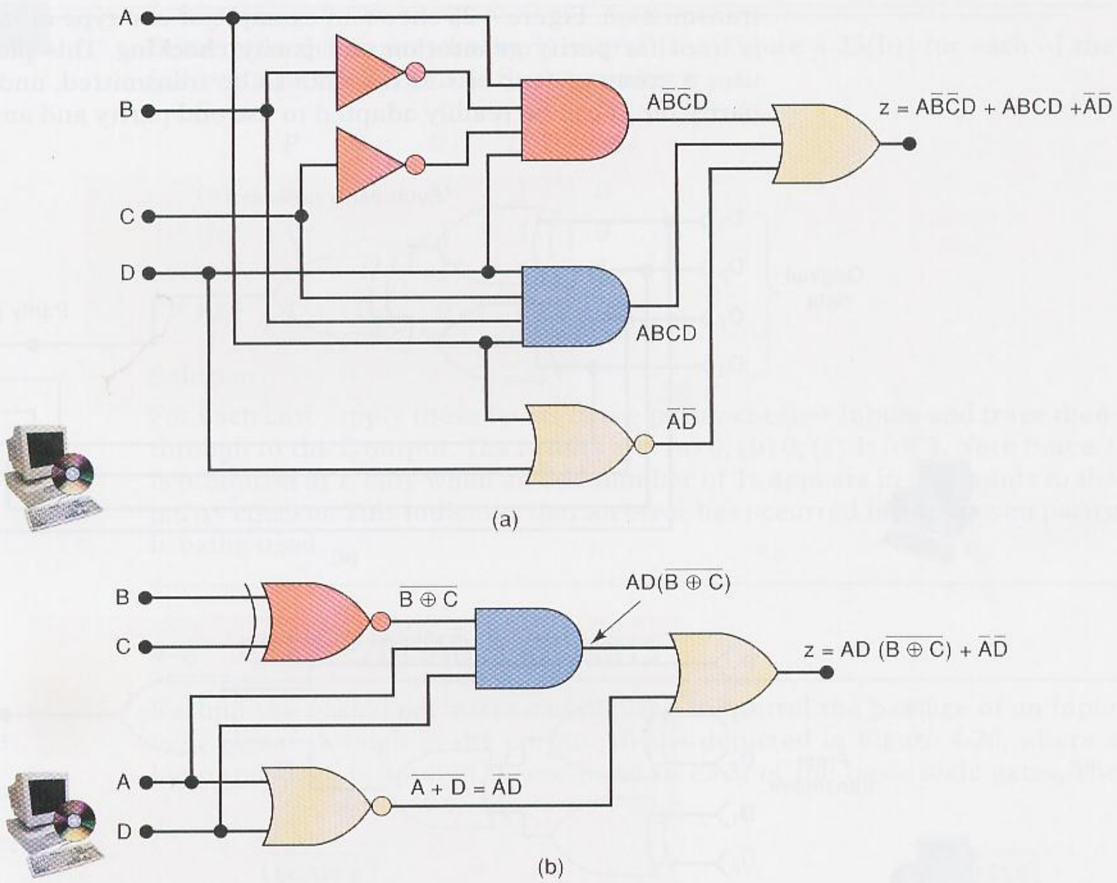

A
B
C
D
$A\overline{B}\,\overline{C}D$
$z = A\overline{B}\,\overline{C}D + ABCD + \overline{A}\,\overline{D}$
ABCD
$\overline{AD}$
(a)
B
C
$\overline{B \oplus C}$
$AD(\overline{B \oplus C})$
$z = AD\,(\overline{B \oplus C}) + \overline{A}\,\overline{D}$
A
D
$\overline{A + D} = \overline{A}\,\overline{D}$
(b)

FIGURE 4-24 Example 4-18, showing how an XNOR gate may be used to simplify circuit implementation.

At first glance, you might think that the expression in parentheses can be replaced by 1. But that would be true only if it were $BC + \overline{B}\,\overline{C}$. You should recognize the expression in parentheses as the XNOR combination of B and C. This fact can be used to reimplement the circuit as shown in Figure 4-24(b). This circuit is much simpler than the original because it uses gates with fewer inputs and two INVERTERs have been eliminated.

REVIEW QUESTIONS

1. Use Boolean algebra to prove that the XNOR output expression is the exact inverse of the XOR output expression.
2. What is the output of an XNOR gate when a logic signal and its exact inverse are connected to its inputs?
3. A logic designer needs an INVERTER, and all that is available is one XOR gate from a 74HC86 chip. Does he need another chip?

4-7 PARITY GENERATOR AND CHECKER

In Chapter 2, we saw that a transmitter can attach a parity bit to a set of data bits before transmitting the data bits to a receiver. We also saw how this allows the receiver to detect any single-bit errors that may have occurred during the transmission. Figure 4-25 shows an example of one type of logic circuitry that is used for **parity generation** and **parity checking**. This particular example uses a group of four bits as the data to be transmitted, and it uses an even-parity bit. It can be readily adapted to use odd parity and any number of bits.

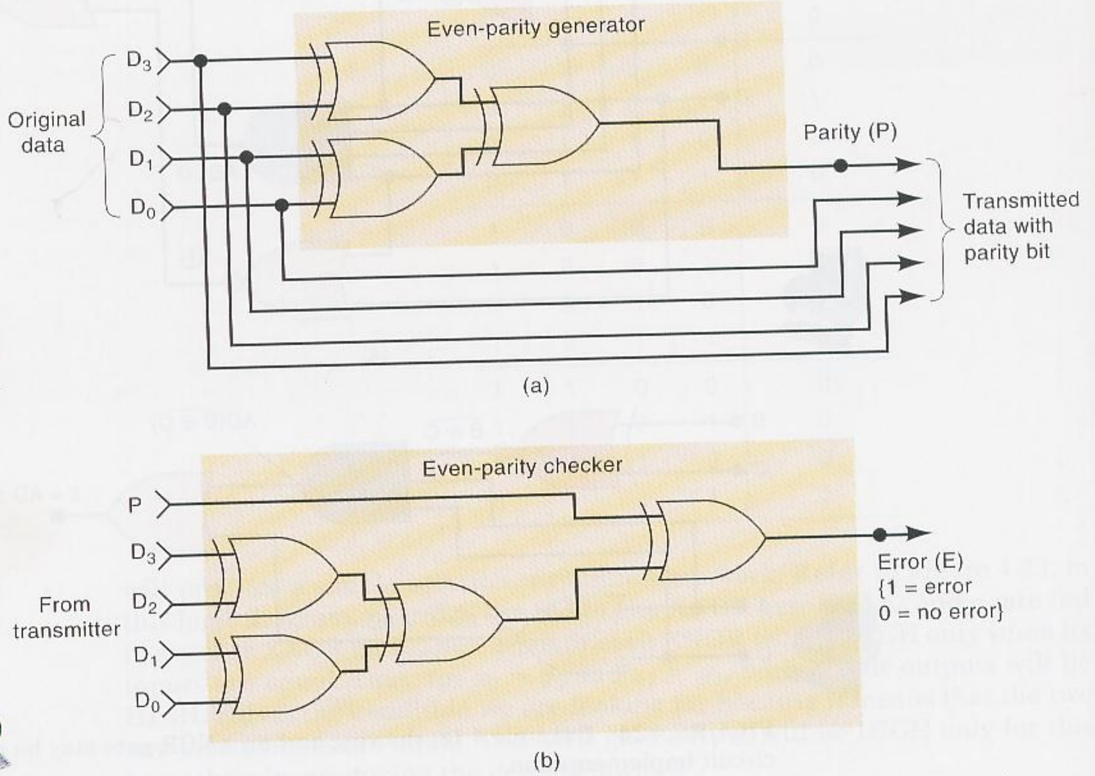


FIGURE 4-25 XOR gates used to implement (a) the parity generator and (b) the parity checker for an even-parity system.

In Figure 4-25(a), the set of data to be transmitted is applied to the parity-generator circuit, which produces the even-parity bit, P, at its output. This parity bit is transmitted to the receiver along with the original data bits, making a total of five bits. In Figure 4-25(b), these five bits (data + parity) enter the receiver's parity-checker circuit, which produces an error output, E, that indicates whether or not a single-bit error has occurred.

It should not be too surprising that both of these circuits employ XOR gates when we consider that a single XOR gate operates so that it produces a 1 output if an odd number of its inputs are 1, and a 0 output if an even number of its inputs are 1.

EXAMPLE 4-19

Determine the parity generator's output for each of the following sets of input data, $D_3D_2D_1D_0$: (a) 0111; (b) 1001; (c) 0000; (d) 0100. Refer to Figure 4-25(a).

Solution

For each case, apply the data levels to the parity-generator inputs and trace them through each gate to the P output. The results are: (a) 1; (b) 0; (c) 0; and (d) 1. Note that P is a 1 only when the original data contain an odd number of 1s. Thus, the total number of 1s sent to the receiver (data + parity) will be even.

EXAMPLE 4-20

Determine the parity checker's output [see Figure 4-25(b)] for each of the following sets of data from the transmitter:

	P	D_3	D_2	D_1	D_0
(a)	0	1	0	1	0
(b)	1	1	1	1	0
(c)	1	1	1	1	1
(d)	1	0	0	0	0

Solution

For each case, apply these levels to the parity-checker inputs and trace them through to the E output. The results are: (a) 0; (b) 0; (c) 1; (d) 1. Note that a 1 is produced at E only when an odd number of 1s appears in the inputs to the parity checker. This indicates that an error has occurred because even parity is being used.

4-8 ENABLE/DISABLE CIRCUITS

Each of the basic logic gates can be used to control the passage of an input logic signal through to the output. This is depicted in Figure 4-26, where a logic signal, A, is applied to one input of each of the basic logic gates. The

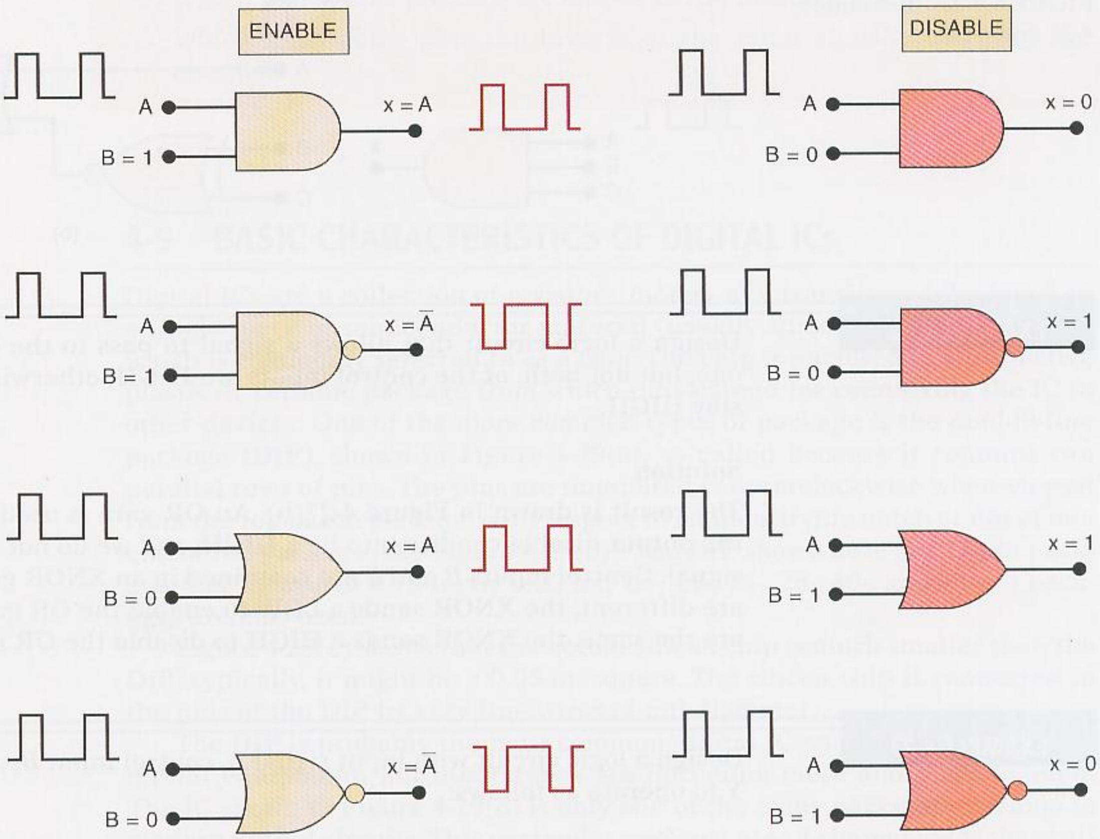


FIGURE 4-26 Four basic gates can either enable or disable the passage of an input signal, A, under control of the logic level at control input B.

other input of each gate is the control input, B. The logic level at this control input will determine whether the input signal is **enabled** to reach the output or **disabled** from reaching the output. This controlling action is why these circuits came to be called *gates*.

Examine Figure 4-26 and you should notice that when the noninverting gates (AND, OR) are enabled, the output will follow the A signal exactly. Conversely, when the inverting gates (NAND, NOR) are enabled, the output will be the exact inverse of the A signal.

Also notice in the figure that AND and NOR gates produce a constant LOW output when they are in the disabled condition. Conversely, the NAND and OR gates produce a constant HIGH output in the disabled condition.

There will be many situations in digital-circuit design where the passage of a logic signal is to be enabled or disabled, depending on conditions present at one or more control inputs. Several are shown in the following examples.

EXAMPLE 4-21

Design a logic circuit that will allow a signal to pass to the output only when control inputs B and C are both HIGH; otherwise, the output will stay LOW.

Solution

An AND gate should be used because the signal is to be passed without inversion, and the disable output condition is a LOW. Because the enable condition must occur only when $B = C = 1$, a three-input AND gate is used, as shown in Figure 4-27(a).

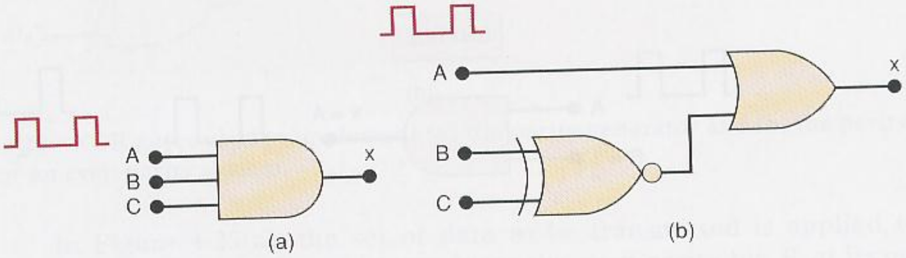


FIGURE 4-27 Examples 4-21 and 4-22.

EXAMPLE 4-22

Design a logic circuit that allows a signal to pass to the output only when one, but not both, of the control inputs are HIGH; otherwise, the output will stay HIGH.

Solution

The result is drawn in Figure 4-27(b). An OR gate is used because we want the output disable condition to be a HIGH, and we do not want to invert the signal. Control inputs B and C are combined in an XNOR gate. When B and C are different, the XNOR sends a LOW to enable the OR gate. When B and C are the same, the XNOR sends a HIGH to disable the OR gate.

EXAMPLE 4-23

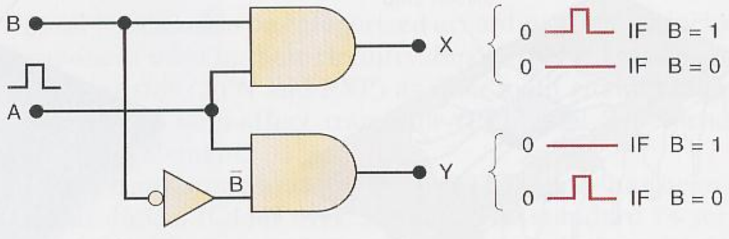
Design a logic circuit with input signal A, control input B, and outputs X and Y to operate as follows:

1. When $B = 1$, output X will follow input A, and output Y will be 0.
2. When $B = 0$, output X will be 0, and output Y will follow input A.

Solution

The two outputs will be 0 when they are disabled and will follow the input signal when they are enabled. Thus, an AND gate should be used for each output. Because X is to be enabled when $B = 1$, its AND gate must be controlled by B, as shown in Figure 4-28. Because Y is to be enabled when $B = 0$, its AND gate is controlled by $\overline{B}$. The circuit in Figure 4-28 is called a *pulse-steering circuit* because it steers the input pulse to one output or the other, depending on B.

FIGURE 4-28 Example 4-23.



REVIEW QUESTIONS

1. Design a logic circuit with three inputs A, B, C and an output that goes LOW only when A is HIGH while B and C are different.
2. Which logic gates produce a 1 output in the disabled state?
3. Which logic gates pass the inverse of the input signal when they are enabled?

4-9 BASIC CHARACTERISTICS OF DIGITAL ICs

Digital ICs are a collection of resistors, diodes, and transistors fabricated on a single piece of semiconductor material (usually silicon) called a *substrate*, which is commonly referred to as a *chip*. The chip is enclosed in a protective plastic or ceramic package from which pins extend for connecting the IC to other devices. One of the more common types of package is the **dual-in-line package (DIP)**, shown in Figure 4-29(a), so called because it contains two parallel rows of pins. The pins are numbered counterclockwise when viewed from the top of the package with respect to an identifying notch or dot at one end of the package [see Figure 4-29(b)]. The DIP shown here is a 14-pin package that measures 0.75 in. by 0.25 in.; 16-, 20-, 24-, 28-, 40-, and 64-pin packages are also used.

Figure 4-29(c) shows that the actual silicon chip is much smaller than the DIP; typically, it might be a 0.05-in. square. The silicon chip is connected to the pins of the DIP by very fine wires (1-mil diameter).

The DIP is probably the most common digital IC package found in older digital equipment, but other types are becoming more and more popular. The IC shown in Figure 4-29(d) is only one of the many packages common to modern digital circuits. This particular package uses J-shaped leads that curl under the IC. We will take a look at some of these other types of IC packages in Chapter 8.

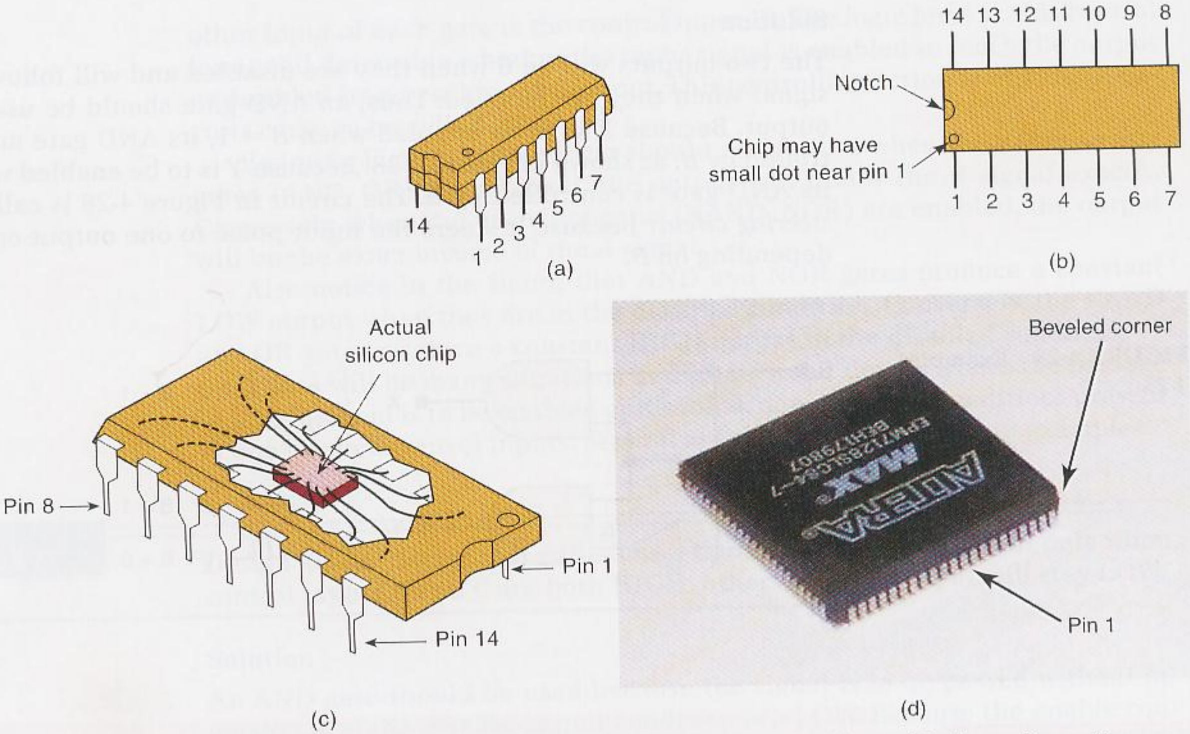


FIGURE 4-29 (a) Dual-in-line package (DIP); (b) top view; (c) actual silicon chip is much smaller than the protective package; (d) PLCC package.

Digital ICs are often categorized according to their circuit complexity as measured by the number of equivalent logic gates on the substrate. There are currently six levels of complexity that are commonly defined as shown in Table 4-5.

TABLE 4-5

Complexity	Gates per Chip
Small-scale integration (SSI)	Fewer than 12
Medium-scale integration (MSI)	12 to 99
Large-scale integration (LSI)	100 to 9999
Very large-scale integration (VLSI)	10,000 to 99,999
Ultra large-scale integration (ULSI)	100,000 to 999,999
Giga-scale integration (GSI)	1,000,000 or more

All of the specific ICs referred to in Chapter 3 and this chapter are SSI chips having a small number of gates. In modern digital systems, medium-scale integration **(MSI)** and large-scale integration devices **(LSI, VLSI, ULSI, GSI)** perform most of the functions that once required several circuit boards full of SSI devices. However, SSI chips are still used as the "interface," or "glue," between these more complex chips. The small-scale ICs also offer an excellent way to learn the basic building blocks of digital systems. Consequently, many laboratory-based courses use these ICs to build and test small projects.

The industrial world of digital electronics has now turned to programmable logic devices (PLDs) to implement a digital system of any significant size. Some simple PLDs are available in DIP packages, but the more complex programmable logic devices require many more pins than are available in DIPs. Larger integrated circuits that may need to be removed from a circuit and replaced are typically manufactured in a plastic leaded chip carrier (PLCC) package. Figure 4-29(d) shows the Altera EPM 7128SLC84 in a PLCC package, which is a very popular PLD used in many educational laboratories. The key features of this chip are more pins, closer spacing, and pins around the entire periphery. Notice that pin 1 is not "on the corner" like the DIP but rather at the middle of the top of the package.

Bipolar and Unipolar Digital ICs

Digital ICs can also be categorized according to the principal type of electronic component used in their circuitry. *Bipolar ICs* are made using the bipolar junction transistor (NPN and PNP) as their main circuit element. *Unipolar ICs* use the unipolar field-effect transistor (P-channel and N-channel MOSFETs) as their main element.

The **transistor-transistor logic (TTL)** family has been the major family of bipolar digital ICs for over 30 years. The standard 74 series was the first series of TTL ICs. It is no longer used in new designs, having been replaced by several higher-performance TTL series, but its basic circuit arrangement forms the foundation for all the TTL series ICs. This circuit arrangement is shown in Figure 4-30(a) for the standard TTL INVERTER. Notice that the circuit contains several bipolar transistors as the main circuit element.

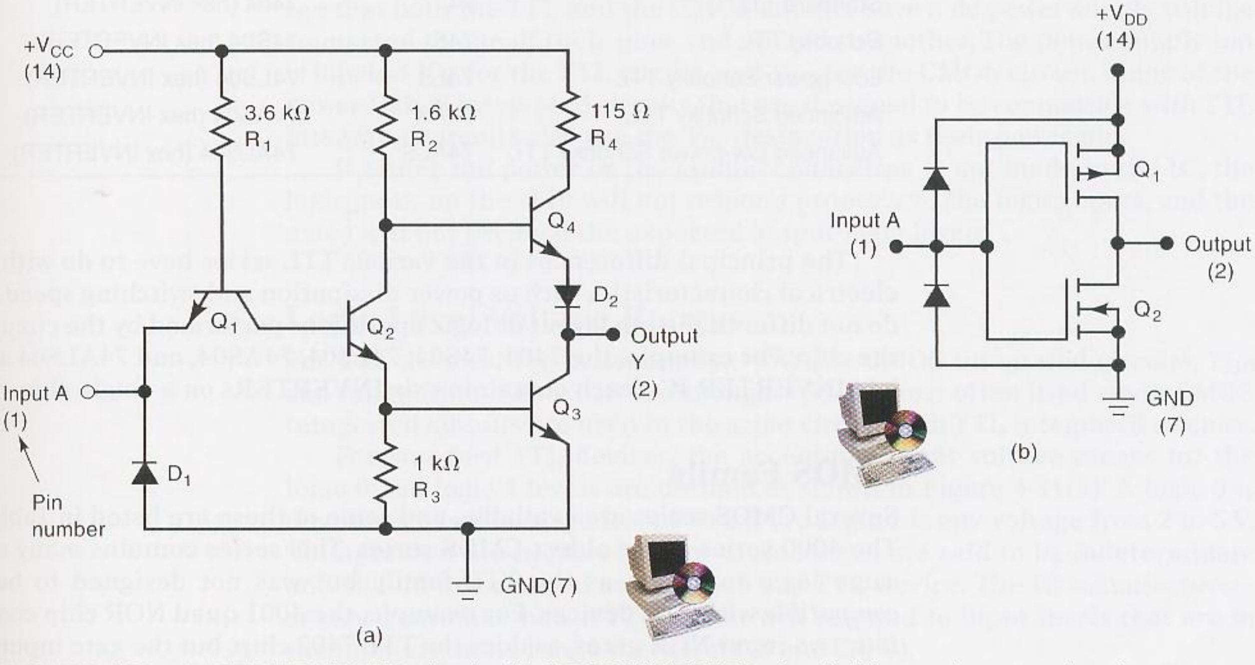


FIGURE 4-30 (a) TTL INVERTER circuit; (b) CMOS INVERTER circuit. Pin numbers are given in parentheses.

TTL had been the leading IC family in the SSI and MSI categories up until the last 12 or so years. Since then, its leading position has been challenged by the CMOS family, which has gradually displaced TTL from that position. The **complementary metal-oxide semiconductor (CMOS)** family belongs to the class of unipolar digital ICs because it uses P- and N-channel MOSFETs as the main circuit elements. Figure 4-30(b) is a standard CMOS INVERTER circuit. If we compare the TTL and CMOS circuits in Figure 4-30, it is apparent that the CMOS version uses fewer components. This is one of the main advantages of CMOS over TTL.

Because of the simplicity and compactness as well as some other superior attributes of CMOS, the modern large-scale ICs are manufactured primarily using CMOS technology. Teaching laboratories that use SSI and MSI devices often use TTL due to its durability, although some use CMOS as well. Chapter 8 will provide a comprehensive study of the circuitry and characteristics of TTL and CMOS. For now, we need to look at only a few of their basic characteristics so that we can talk about troubleshooting simple combinational circuits.

TTL Family

The TTL logic family actually consists of several subfamilies or series. Table 4-6 lists the name of each TTL series together with the prefix designation used to identify different ICs as being part of that series. For example, ICs that are part of the standard TTL series have an identification number that starts with 74. The 7402, 7438, and 74123 are all ICs in this series. Likewise, ICs that are part of the low-power Schottky TTL series have an identification number that starts with 74LS. The 74LS02, 74LS38, and 74LS123 are examples of devices in the 74LS series.

TABLE 4-6 Various series within the TTL logic family.

TTL Series	Prefix	Example IC
Standard TTL	74	7404 (hex INVERTER)
Schottky TTL	74S	74S04 (hex INVERTER)
Low-power Schottky TTL	74LS	74LS04 (hex INVERTER)
Advanced Schottky TTL	74AS	74AS04 (hex INVERTER)
Advanced low-power Schottky TTL	74ALS	74ALS04 (hex INVERTER)

The principal differences in the various TTL series have to do with their electrical characteristics such as power dissipation and switching speed. They do not differ in the pin layout or logic operations performed by the circuits on the chip. For example, the 7404, 74S04, 74LS04, 74AS04, and 74ALS04 are all hex-INVERTER ICs, each containing *six* INVERTERs on a single chip.

CMOS Family

Several CMOS series are available, and some of these are listed in Table 4-7. The 4000 series is the oldest CMOS series. This series contains many of the same logic functions as the TTL family but was not designed to be *pin-compatible* with TTL devices. For example, the 4001 quad NOR chip contains four two-input NOR gates, as does the TTL 7402 chip, but the gate inputs and outputs on the CMOS chip will not have the same pin numbers as the corresponding signals on the TTL chip.

The 74C, 74HC, 74HCT, 74AC, and 74ACT series are newer CMOS series. The first three are pin-compatible with correspondingly numbered TTL devices. For example, the 74C02, 74HC02, and 74HCT02 have the same pin layout as the 7402, 74LS02, and so on. The 74HC and 74HCT series operate at a higher speed than 74C devices. The 74HCT series is designed to be *electrically compatible* with TTL devices; that is, a 74HCT integrated circuit can be connected directly to TTL devices without any interfacing circuitry. The 74AC and 74ACT series are advanced-performance ICs. Neither is pin-compatible with

TABLE 4-7 Various series within the CMOS logic family.

CMOS Series	Prefix	Example IC
Metal-gate CMOS	40	4001 (quad NOR gates)
Metal-gate, pin-compatible with TTL	74C	74C02 (quad NOR gates)
Silicon-gate, pin-compatible with TTL, high-speed	74HC	74HC02 (quad NOR gates)
Silicon-gate, high-speed, pin-compatible and electrically compatible with TTL	74HCT	74HCT02 (quad NOR gates)
Advanced-performance CMOS, not pin-compatible or electrically compatible with TTL	74AC	74AC02 (quad NOR)
Advanced-performance CMOS, not pin-compatible with TTL, but electrically compatible with TTL	74ACT	74ACT02 (quad NOR)

TTL. The 74ACT devices are electrically compatible with TTL. We explore the various TTL and CMOS series in greater detail in Chapter 8.

Power and Ground

To use digital ICs, it is necessary to make the proper connections to the IC pins. The most important connections are *dc power* and *ground.* These are required for the circuits on the chip to operate correctly. In Figure 4-30, you can see that both the TTL and the CMOS circuits have a dc power supply voltage connected to one of their pins, and ground to another. The power supply pin is labeled V_{CC} for the TTL circuit, and V_{DD} for the CMOS circuit. Many of the newer CMOS integrated circuits that are designed to be compatible with TTL integrated circuits also use the V_{CC} designation as their power pin.

If either the power or the ground connection is not made to the IC, the logic gates on the chip will not respond properly to the logic inputs, and the gates will not produce the expected output logic levels.

Logic-Level Voltage Ranges

For TTL devices, V_{CC} is nominally +5 V. For CMOS integrated circuits, V_{DD} can range from +3 to +18 V, although +5 V is most often used when CMOS integrated circuits are used in the same circuit with TTL integrated circuits.

For standard TTL devices, the acceptable input voltage ranges for the logic 0 and logic 1 levels are defined as shown in Figure 4-31(a). A logic 0 is any voltage in the range from 0 to 0.8 V; a logic 1 is any voltage from 2 to 5 V. Voltages that are not in either of these ranges are said to be **indeterminate** and should not be used as inputs to any TTL device. The IC manufacturers cannot guarantee how a TTL circuit will respond to input levels that are in the indeterminate range (between 0.8 and 2.0 V).

The logic input voltage ranges for CMOS integrated circuits operating with $V_{DD} = +5$ V are shown in Figure 4-31(b). Voltages between 0 and 1.5 V are defined as a logic 0, and voltages from 3.5 to 5 V are defined as a logic 1. The indeterminate range includes voltages between 1.5 and 3.5 V.

Unconnected (Floating) Inputs

What happens when the input to a digital IC is left unconnected? An unconnected input is often called a **floating** input. The answer to this question will be different for TTL and CMOS.

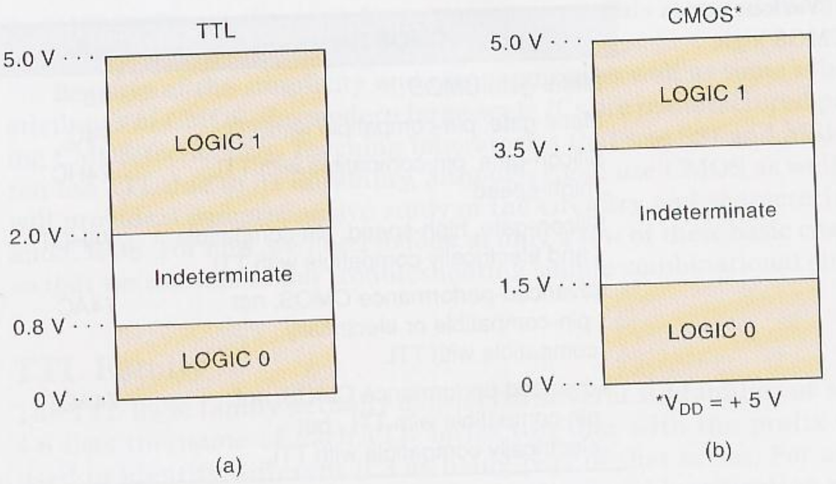


FIGURE 4-31 Logic-level input voltage ranges for (a) TTL and (b) CMOS digital ICs.

A floating TTL input acts just like a logic 1. In other words, the IC will respond as if the input had a logic HIGH level applied to it. This characteristic is often used when testing a TTL circuit. A lazy technician might leave certain inputs unconnected instead of connecting them to a logic HIGH. Although this is logically correct, it is not a recommended practice, especially in final circuit designs, because the floating TTL input is extremely susceptible to picking up noise signals that will probably adversely affect the device's operation.

A floating input on some TTL gates will measure a dc level of between 1.4 and 1.8 V when checked with a VOM or an oscilloscope. Even though this is in the indeterminate range for TTL, it will produce the same response as a logic 1. Being aware of this characteristic of a floating TTL input can be valuable when troubleshooting TTL circuits.

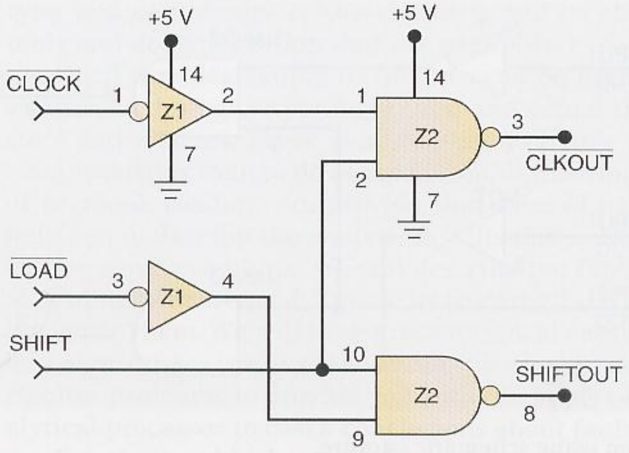
If a CMOS input is left floating, it may have disastrous results. The IC may become overheated and eventually destroy itself. For this reason all inputs to a CMOS integrated circuit must be connected to a LOW or a HIGH level or to the output of another IC. A floating CMOS input will not measure as a specific dc voltage but will fluctuate randomly as it picks up noise. Thus, it does not act as logic 1 or logic 0, and so its effect on the output is unpredictable. Sometimes the output will oscillate as a result of the noise picked up by the floating input.

Many of the more complex CMOS ICs have circuitry built into the inputs, which reduces the likelihood of any destructive reaction to an open input. With this circuitry, it is not necessary to ground each unused pin on these large ICs when experimenting. It is still good practice, however, to tie unused inputs to HIGH or LOW (whichever is appropriate) in the final circuit implementation.

Logic-Circuit Connection Diagrams

A connection diagram shows *all* electrical connections, pin numbers, IC numbers, component values, signal names, and power supply voltages. Figure 4-32 shows a typical connection diagram for a simple logic circuit. Examine it carefully and note the following important points:

1. The circuit uses logic gates from two different ICs. The two INVERTERs are part of a 74HC04 chip that has been given the designation Z1. The 74HC04 contains six INVERTERs; two of them are used in this circuit, and each is labeled as being part of chip Z1. Similarly, the two NAND gates are part of a 74HC00 chip that contains four NAND gates. All of



IC	Type
Z1	74HC04 hex inverter
Z2	74HC00 quad nand

FIGURE 4-32 Typical logic-circuit connection diagram.

the gates on this chip are designated with the label Z2. By numbering each gate as Z1, Z2, Z3, and so on, we can keep track of which gate is part of which chip. This is especially valuable in more complex circuits containing many ICs with several gates per chip.

2. Each gate input and output pin number is indicated on the diagram. These pin numbers and the IC labels are used to reference easily any point in the circuit. For example, Z1 pin 2 refers to the output pin of the top INVERTER. Similarly, we can say that Z1 pin 4 is connected to Z2 pin 9.
3. The power and ground connections to each IC (not each gate) are shown on the diagram. For example, Z1 pin 14 is connected to +5 V, and Z1 pin 7 is connected to ground. These connections provide power to *all* of the six INVERTERs that are part of Z1.
4. For the circuit contained in Figure 4-32, the signals that are inputs are on the left. The signals that are outputs are on the right. The bar over the signal name indicates that the signal is active when LOW. The bubbles are positioned on the diagram symbols also to indicate the active-LOW state. Each signal in this case is obviously a single bit.
5. Signals are defined graphically in Figure 4-32 as inputs and outputs, and the relationship between them (the operation of the circuit) is described graphically using interconnected logic symbols.

Manufacturers of electronic equipment generally supply detailed schematics that use a format similar to that in Figure 4-32. These connection diagrams are a virtual necessity when troubleshooting a faulty circuit. We have chosen to identify individual ICs as Z1, Z2, Z3, and so on. Other designations that are commonly used are IC1, IC2, IC3, and so on, and U1, U2, U3, and so on.

Personal computers with schematic diagram software can be used to draw logic circuits. Computer applications that can interpret these graphic symbols and signal connections and can translate them into logical relationships are often called schematic capture tools. The Altera MAX+PLUS development system for programmable logic allows the user to enter graphic design files (.gdf) using schematic capture techniques. Thus, designing the circuit is as easy as drawing the schematic diagram on the computer screen. Notice that in Figure 4-33 there are no pin numbers or chip designations on the logic symbols. The circuits will not be implemented using actual SSI or MSI chips, but rather the equivalent logic functionality will be "programmed" into a PLD. We will explain this further at a later point in this chapter.

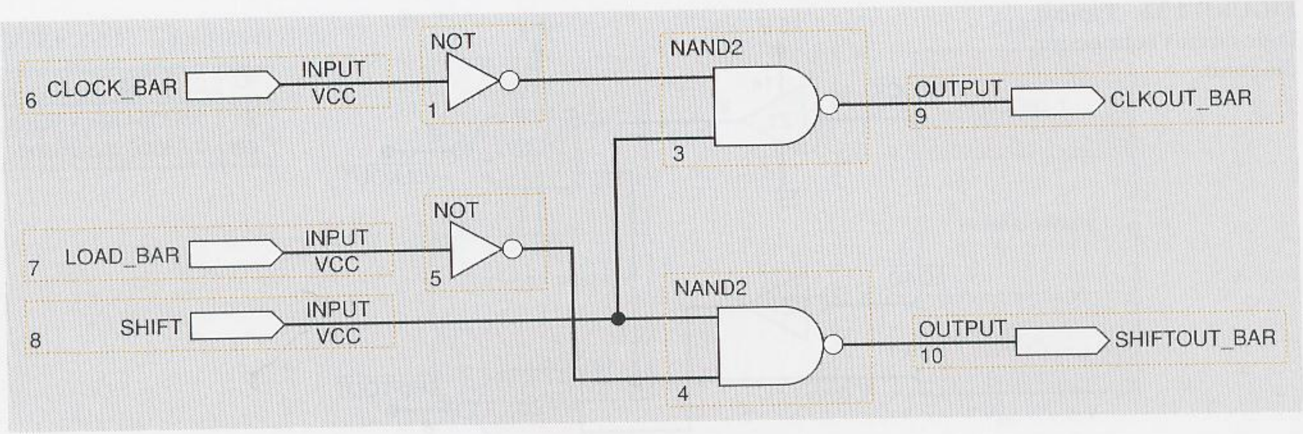


FIGURE 4-33 Logic diagram using schematic capture.

REVIEW QUESTIONS

1. What is the most common type of digital IC package?
2. Name the six common categories of digital ICs according to complexity.
3. *True or false:* A 74S74 chip will contain the same logic and pin layout as the 74LS74.
4. *True or false:* A 74HC74 chip will contain the same logic and pin layout as the 74AS74.
5. Which CMOS series are not pin-compatible with TTL?
6. What is the acceptable input voltage range of a logic 0 for TTL? What is it for a logic 1?
7. Repeat question 6 for CMOS operating at $V_{DD} = 5$ V.
8. How does a TTL integrated circuit respond to a floating input?
9. How does a CMOS integrated circuit respond to a floating input?
10. Which CMOS series can be connected directly to TTL with no interfacing circuitry?
11. What is the purpose of pin numbers on a logic circuit connection diagram?
12. What are the key similarities of graphic design files used for programmable logic and traditional logic circuit connection diagrams?

4-10 TROUBLESHOOTING DIGITAL SYSTEMS

There are three basic steps in fixing a digital circuit or system that has a fault (failure):

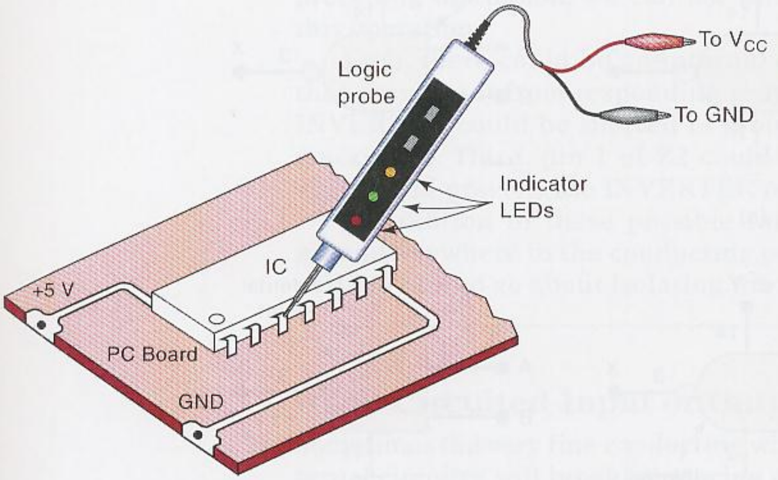
1. *Fault detection.* Observe the circuit/system operation and compare it with the expected correct operation.
2. *Fault isolation.* Perform tests and make measurements to isolate the fault.
3. *Fault correction.* Replace the faulty component, repair the faulty connection, remove the short, and so on.

Although these steps may seem relatively apparent and straightforward, the actual troubleshooting procedure that is followed is highly dependent on the type and complexity of the circuitry, and on the kinds of troubleshooting tools and documentation that are available.

Good troubleshooting techniques can be learned only in a laboratory environment through experimentation and actual troubleshooting of faulty circuits and systems. There is absolutely no better way to become an effective troubleshooter than to do as much troubleshooting as possible, and no amount of textbook reading can provide that kind of experience. We can, however, help you to develop the analytical skills that are the most essential part of effective troubleshooting. We will describe the types of faults that are common to systems that are made primarily from digital ICs and we will tell you how to recognize them. We will then present typical case studies to illustrate the analytical processes involved in troubleshooting. In addition, there will be end-of-chapter problems to provide you with the opportunity to go through these analytical processes to reach conclusions about faulty digital circuits.

For the troubleshooting discussions and exercises we will be doing in this book, it will be assumed that the troubleshooting technician has the basic troubleshooting tools available: *logic probe, oscilloscope, logic pulser.* Of course, the most important and effective troubleshooting tool is the technician's brain, and that's the tool we are hoping to develop by presenting troubleshooting principles and techniques, examples and problems, here and in the following chapters.

In the next three sections on troubleshooting, we will use only our brain and a **logic probe** such as the one illustrated in Figure 4-34. The logic probe has a pointy metal tip that is touched to the specific point you want to test. Here, it is shown probing (touching) pin 3 of an IC. It can also be touched to a printed circuit board trace, an uninsulated wire, a connector pin, a lead on a discrete component such as a transistor, or any other conducting point in a circuit. The logic level that is present at the probe tip will be indicated by the status of the indicator LEDs in the probe. The four possibilities are given in the table of Figure 4-34. Note that an *indeterminate* logic level produces no indicator light. This includes the condition where the probe tip is touched to a point in a circuit that is open or floating—that is, not connected to any source of voltage. This type of probe also offers a yellow LED to indicate the presence of a pulse train. Any transitions (LOW to HIGH or HIGH to LOW) will cause the yellow LED to flash on for a fraction of a second and then turn off. If the transitions are occurring frequently, the LED will continue to flash



LEDs			
Red	Green	Yellow	Logic Condition
OFF	ON	OFF	LOW
ON	OFF	OFF	HIGH
OFF	OFF	OFF	INDETERMINATE*
X	X	FLASHING	PULSING

* Includes open or floating condition

FIGURE 4-34 A logic probe is used to monitor the logic level activity at an IC pin or any other accessible point in a logic circuit.

at around 3 Hz. By observing the green and red LEDs along with the flashing yellow, you can tell whether the signal is mostly HIGH or mostly LOW.

4-11 INTERNAL DIGITAL IC FAULTS

The most common internal failures of digital ICs are:

1. Malfunction in the internal circuitry
2. Inputs or outputs shorted to ground or V_{CC}
3. Inputs or outputs open-circuited
4. Short between two pins (other than ground or V_{CC})

We will now describe each of these types of failure.

Malfunction in Internal Circuitry

This is usually caused by one of the internal components failing completely or operating outside its specifications. When this happens, the IC outputs do not respond properly to the IC inputs. There is no way to predict what the outputs will do because it depends on what internal component has failed. Examples of this type of failure would be a base-emitter short in transistor Q_4 or an extremely large resistance value for R_2 in the TTL INVERTER of Figure 4-30(a). This type of internal IC failure is not as common as the other three.

Input Internally Shorted to Ground or Supply

This type of internal failure will cause an IC input to be stuck in the LOW or HIGH state. Figure 4-35(a) shows input pin 2 of a NAND gate shorted to ground within the IC. This will cause pin 2 always to be in the LOW state. If this input pin is being driven by a logic signal B, it will effectively short B to ground. Thus, this type of fault will affect the output of the device that is generating the B signal.

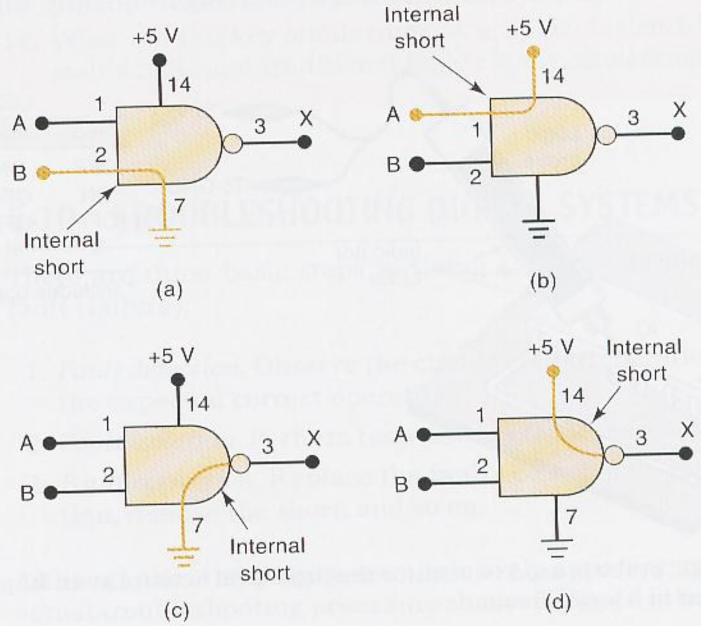


FIGURE 4-35 (a) IC input internally shorted to ground; (b) IC input internally shorted to supply voltage. These two types of failures force the input signal at the shorted pin to stay in the same state. (c) IC output internally shorted to ground; (d) output internally shorted to supply voltage. These two failures do not affect signals at the IC inputs.

Similarly, an IC input pin could be internally shorted to +5 V, as in Figure 4-35(b). This would keep that pin stuck in the HIGH state. If this input pin is being driven by a logic signal *A*, it will effectively short *A* to +5 V.

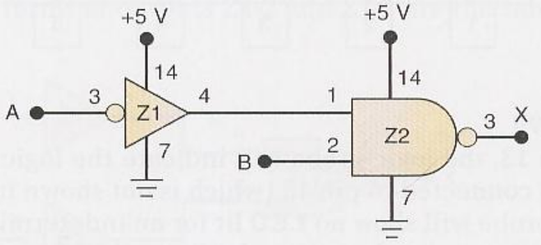
Output Internally Shorted to Ground or Supply

This type of internal failure will cause the output pin to be stuck in the LOW or HIGH state. Figure 4-35(c) shows pin 3 of the NAND gate shorted to ground within the IC. This output is stuck LOW, and it will not respond to the conditions applied to input pins 1 and 2; in other words, logic inputs *A* and *B* will have no effect on output *X*.

An IC output pin can also be shorted to +5 V within the IC, as shown in Figure 4-35(d). This forces the output pin 3 to be stuck HIGH regardless of the state of the signals at the input pins. Note that this type of failure has no effect on the logic signals at the IC inputs.

EXAMPLE 4-24

Refer to the circuit of Figure 4-36. A technician uses a logic probe to determine the conditions at the various IC pins. The results are recorded in the figure. Examine these results and determine if the circuit is working properly. If not, suggest some of the possible faults.



Pin	Condition
Z1-3	Pulsing
Z1-4	LOW
Z2-1	LOW
Z2-2	HIGH
Z2-3	HIGH

FIGURE 4-36 Example 4-24.

Solution

Output pin 4 of the INVERTER should be pulsing because its input is pulsing. The recorded results, however, show that pin 4 is stuck LOW. Because this is connected to Z2 pin 1, this keeps the NAND output HIGH. From our preceding discussion, we can list three possible faults that could produce this operation.

First, there could be an internal component failure in the INVERTER that prevents it from responding properly to its input. Second, pin 4 of the INVERTER could be shorted to ground internal to Z1, thereby keeping it stuck LOW. Third, pin 1 of Z2 could be shorted to ground internal to Z2. This would prevent the INVERTER output pin from changing.

In addition to these possible faults, there can be external shorts to ground anywhere in the conducting path between Z1 pin 4 and Z2 pin 1. We will see how to go about isolating the actual fault in a subsequent example.

Open-Circuited Input or Output

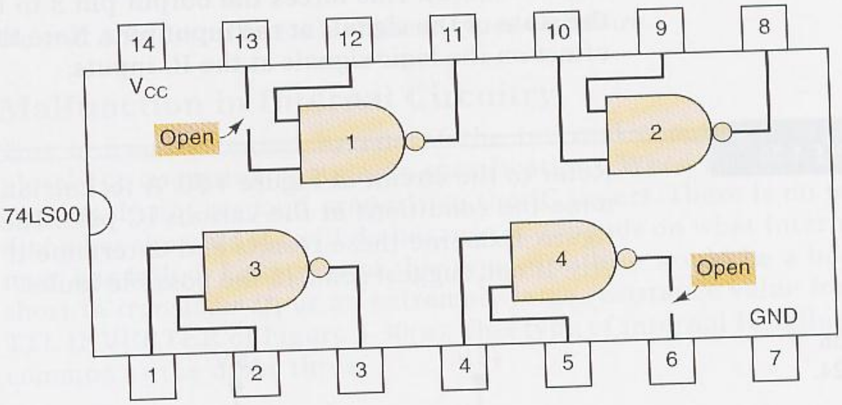
Sometimes the very fine conducting wire that connects an IC pin to the IC's internal circuitry will break, producing an open circuit. Figure 4-37 in Example 4-25 shows this for an input (pin 13) and an output (pin 6). If a signal is applied to pin 13, it will not reach the NAND-1 gate input and so will not have an effect on the NAND-1 output. The open gate input will be in the floating state. As stated earlier, TTL devices will respond as if this floating input is a logic 1, and CMOS devices will respond erratically and may even become damaged from overheating.

The open at the NAND-4 output prevents the signal from reaching IC pin 6, so there will be no stable voltage present at that pin. If this pin is connected to the input of another IC, it will produce a floating condition at that input.

EXAMPLE 4-25

What would a logic probe indicate at pin 13 and at pin 6 of Figure 4-37?

FIGURE 4-37 An IC with an internally open input will not respond to signals applied to that input pin. An internally open output will produce an unpredictable voltage at that output pin.



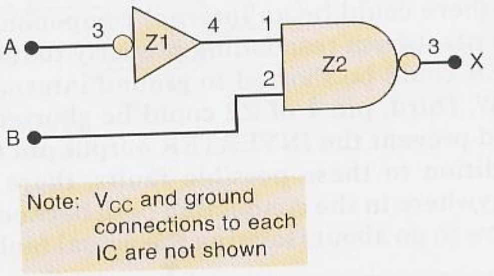
Solution

At pin 13, the logic probe will indicate the logic level of the external signal that is connected to pin 13 (which is not shown in the diagram). At pin 6, the logic probe will show no LED lit for an indeterminate logic level because the NAND output level never makes it to pin 6.

EXAMPLE 4-26

Refer to the circuit of Figure 4-38 and the recorded logic probe indications. What are some of the possible faults that could produce the recorded results? Assume that the ICs are TTL.

FIGURE 4-38 Example 4-26.



Pin	Condition
Z1-3	HIGH
Z1-4	LOW
Z2-1	LOW
Z2-2	Pulsing
Z2-3	Pulsing

Note: V_{CC} and ground connections to each IC are not shown

Solution

Examination of the recorded results indicates that the INVERTER appears to be working properly, but the NAND output is inconsistent with its inputs. The NAND output should be HIGH because its input pin 1 is LOW. This LOW should prevent the NAND gate from responding to the pulses at pin 2. It is probable that this LOW is not reaching the internal NAND gate circuitry because of an internal open. Because the IC is TTL, this open circuit would produce the same effect as a logic HIGH at pin 1. If the IC had been CMOS, the internal open circuit at pin 1 might have produced an indeterminate output and possible overheating and destruction of the chip.

From our earlier statement regarding open TTL inputs, you might have expected that the voltage of pin 1 of Z2 would be 1.4 to 1.8 V and should have been registered as indeterminate by the logic probe. This would have been true if the open circuit had been *external* to the NAND chip. There is no open circuit between Z1 pin 4 and Z2 pin 1, and so the voltage at Z1 pin 4 is reaching Z2 pin 1, but it becomes disconnected *inside* the NAND chip.

Short Between Two Pins

An internal short between two pins of an IC will force the logic signals at those pins always to be identical. Whenever two signals that are supposed to be different show the same logic-level variations, there is a good possibility that the signals are shorted together.

Consider the circuit in Figure 4-39, where pins 5 and 6 of the NOR gate are internally shorted together. The short causes the two INVERTER output pins to be connected together so that the signals at Z1 pin 2 and Z1 pin 4 must be identical, even when the two INVERTER input signals are trying to produce different outputs. To illustrate, consider the input waveforms shown in the diagram. Even though these input waveforms are different, the waveforms at outputs Z1-2 and Z1-4 are the same.

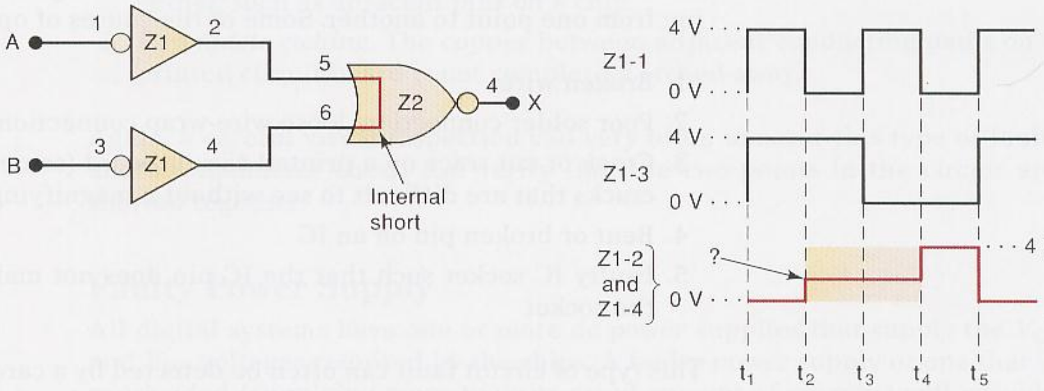


FIGURE 4-39 When two input pins are internally shorted, the signals driving these pins are forced to be identical, and usually a signal with three distinct levels results.

During the interval t_1 to t_2, both INVERTERs have a HIGH input and both are trying to produce a LOW output, so that their being shorted together makes no difference. During the interval t_4 to t_5, both INVERTERs have a LOW input and are trying to produce a HIGH output, so that again their being shorted has no effect. However, during the intervals t_2 to t_3 and t_3 to t_4, one INVERTER is trying to produce a HIGH output while the other is trying to produce a LOW output. This is called signal **contention** because the two signals are "fighting" each other. When this happens, the actual voltage level that appears at the shorted outputs will depend on the internal IC circuitry. For TTL devices, it will usually be a voltage in the high end of the logic 0 range (i.e., close to 0.8 V), although it may also be in the indeterminate range. For CMOS devices, it will often be a voltage in the indeterminate range.

Whenever you see a waveform like the Z1-2, Z1-4 signal in Figure 4-39 with three different levels, you should suspect that two output signals may be shorted together.

REVIEW QUESTIONS

1. List the different internal digital IC faults.
2. Which internal IC fault can produce signals that show three different voltage levels?
3. What would a logic probe indicate at Z1-2 and Z1-4 of Figure 4-39 if $A = 0$ and $B = 1$?
4. What is signal contention?

4-12 EXTERNAL FAULTS

We have seen how to recognize the effects of various faults internal to digital ICs. Many more things can go wrong external to the ICs; we will describe the most common ones in this section.

Open Signal Lines

This category includes any fault that produces a break or discontinuity in the conducting path such that a voltage level or signal is prevented from going from one point to another. Some of the causes of open signal lines are:

1. Broken wire
2. Poor solder connection; loose wire-wrap connection
3. Crack or cut trace on a printed circuit board (some of these are hairline cracks that are difficult to see without a magnifying glass)
4. Bent or broken pin on an IC
5. Faulty IC socket such that the IC pin does not make good contact with the socket

This type of circuit fault can often be detected by a careful visual inspection and then verified by disconnecting power from the circuit and checking for continuity (i.e., a low-resistance path) with an ohmmeter between the two points in question.

EXAMPLE 4-27

Consider the CMOS circuit of Figure 4-40 and the accompanying logic probe indications. What is the most probable circuit fault?

Solution

The indeterminate level at the NOR gate output is probably due to the indeterminate input at pin 2. Because there is a LOW at Z1-6, this LOW should also be at Z2-2. Clearly, the LOW from Z1-6 is not reaching Z2-2, and there must be an open circuit in the signal path between these two points. The location of this open circuit can be determined by starting at Z1-6 with the logic probe and tracing the LOW level along the signal path toward Z2-2 until it changes into an indeterminate level.

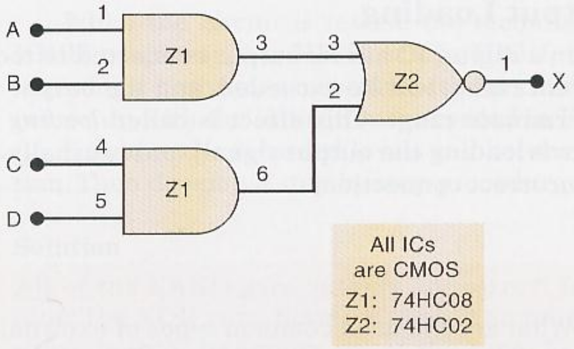


FIGURE 4-40 Example 4-27.

Pin	Condition
Z1-1	Pulsing
Z1-2	HIGH
Z1-3	Pulsing
Z1-4	LOW
Z1-5	Pulsing
Z1-6	LOW
Z2-3	Pulsing
Z2-2	Indeterminate
Z2-1	Indeterminate

Shorted Signal Lines

This type of fault has the same effect as an internal short between IC pins. It causes two signals to be exactly the same (signal contention). A signal line may be shorted to ground or V_{CC} rather than to another signal line. In those cases, the signal will be forced to the LOW or the HIGH state. The main causes for unexpected shorts between two points in a circuit are as follows:

1. *Sloppy wiring.* An example of this is stripping too much insulation from ends of wires that are in close proximity.
2. *Solder bridges.* These are splashes of solder that short two or more points together. They commonly occur between points that are very close together, such as adjacent pins on a chip.
3. *Incomplete etching.* The copper between adjacent conducting paths on a printed circuit board is not completely etched away.

Again, a careful visual inspection can very often uncover this type of fault, and an ohmmeter check can verify that the two points in the circuit are shorted together.

Faulty Power Supply

All digital systems have one or more dc power supplies that supply the V_{CC} and V_{DD} voltages required by the chips. A faulty power supply or one that is overloaded (supplying more than its rated amount of current) will provide poorly regulated supply voltages to the ICs, and the ICs either will not operate or will operate erratically.

A power supply may go out of regulation because of a fault in its internal circuitry, or because the circuits that it is powering are drawing more current than the supply is designed for. This can happen if a chip or a component has a fault that causes it to draw much more current than normal.

It is good troubleshooting practice to check the voltage levels at each power supply in the system to see that they are within their specified ranges. It is also a good idea to check them on an oscilloscope to verify that there is no significant amount of ac ripple on the dc levels and to verify that the voltage levels stay regulated during the system operation.

One of the most common signs of a faulty power supply is one or more chips operating erratically or not at all. Some ICs are more tolerant of power supply variations and may operate properly, while others do not. You should always check the power and ground levels at each IC that appears to be operating incorrectly.

Output Loading

When a digital IC has its output connected to too many IC inputs, its output current rating will be exceeded, and the output voltage can fall into the indeterminate range. This effect is called *loading* the output signal (actually it's overloading the output signal) and is usually the result of poor design or an incorrect connection.

REVIEW QUESTIONS

1. What are the most common types of external faults?
2. List some of the causes of signal-path open circuits.
3. What symptoms are caused by a faulty power supply?
4. How might loading affect an IC output voltage level?

4-13 TROUBLESHOOTING CASE STUDY

The following example will illustrate the analytical processes involved in troubleshooting digital circuits. Although the example is a fairly simple combinational logic circuit, the reasoning and the troubleshooting procedures used can be applied to the more complex digital circuits that we encounter in subsequent chapters.

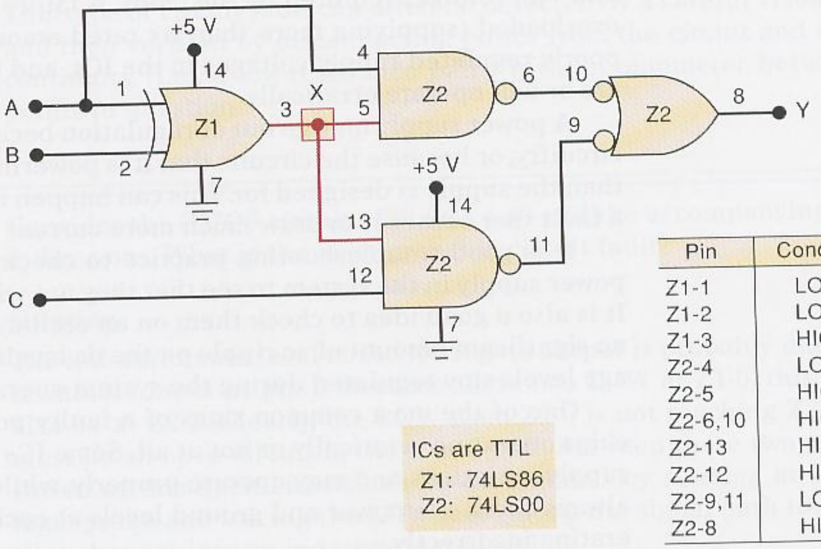
EXAMPLE 4-28

Consider the circuit of Figure 4-41. Output Y is supposed to go HIGH for either of the following conditions:

1. $A = 1, B = 0$ regardless of the level on C
2. $A = 0, B = 1, C = 1$

You may wish to verify these results for yourself.

FIGURE 4-41 Example 4-28.



Pin	Condition
Z1-1	LOW
Z1-2	LOW
Z1-3	HIGH
Z2-4	LOW
Z2-5	HIGH
Z2-6,10	HIGH
Z2-13	HIGH
Z2-12	HIGH
Z2-9,11	LOW
Z2-8	HIGH

When the circuit is tested, the technician observes that output Y goes HIGH whenever A is HIGH or C is HIGH, regardless of the level at B. She takes logic probe measurements for the condition where $A = B = 0, C = 1$ and comes up with the indications recorded in Figure 4-41.

Examine the recorded levels and list the possible causes for the malfunction. Then develop a step-by-step procedure to determine the exact fault.

Solution

All of the NAND gate outputs are correct for the levels present at their inputs. The XOR gate, however, should be producing a LOW at output pin 3 because both of its inputs are at the same LOW level. It appears that Z1-3 is stuck HIGH, even though its inputs should produce a LOW. There are several possible causes for this:

1. An internal component failure in Z1 that prevents its output from going LOW
2. An external short to V_{CC} from any point along the conductors connected to node X (shaded in the diagram of the figure)
3. Pin 3 of Z1 internally shorted to V_{CC}
4. Pin 5 of Z2 internally shorted to V_{CC}
5. Pin 13 of Z2 internally shorted to V_{CC}

All of these possibilities except for the first one will short node X (and every IC pin connected to it) directly to V_{CC}.

The following procedure can be used to isolate the fault. This procedure is not the only approach that can be used and, as we stated earlier, the actual troubleshooting procedure that a technician uses is very dependent on what test equipment is available.

1. Check the V_{CC} and ground levels at the appropriate pins of Z1. Although it is unlikely that the absence of either of these might cause Z1-3 to stay HIGH, it is a good idea to make this check on any IC that is producing an incorrect output.
2. Turn off power to the circuit and use an ohmmeter to check for a short (resistance less than 1 Ω) between node X and any point connected to V_{CC} (such as Z1-14 or Z2-14). If no short is indicated, the last four possibilities in our list can be eliminated. This means that it is very likely that Z1 has an internal failure and should be replaced.
3. If step 2 shows that there is a short from node X to V_{CC}, perform a thorough visual examination of the circuit board and look for solder bridges, unetched copper slivers, uninsulated wires touching each other, and any other possible cause of an external short to V_{CC}. A likely spot for a solder bridge would be between adjacent pins 13 and 14 of Z2. Pin 14 is connected to V_{CC}, and pin 13 to node X. If an external short is found, remove it and perform an ohmmeter check to verify that node X is no longer shorted to V_{CC}.
4. If step 3 does not reveal an external short, the three possibilities that remain are internal shorts to V_{CC} at Z1-3, Z2-13, or Z2-5. One of these is shorting node X to V_{CC}.

To determine which of these IC pins is the culprit, we should disconnect each of them from node X *one at a time* and recheck for a short to V_{CC} after each disconnection. When the pin that is internally shorted to V_{CC} is disconnected, node X will no longer be shorted to V_{CC}.

The process of disconnecting each suspected pin from node *X* can be easy or difficult depending on how the circuit is constructed. If the ICs are in sockets, all you need to do is to pull the IC from its socket, bend out the suspected pin, and reinsert the IC into its socket. If the ICs are soldered into a printed circuit board, you will have to cut the trace that is connected to the pin and repair the cut trace when you are finished.

Example 4-28, although fairly simple, shows you the kinds of thinking that a troubleshooter must employ to isolate a fault. You will have the opportunity to begin developing your own troubleshooting skills by working on many end-of-chapter problems that have been designated with a **T** for troubleshooting.

4-14 PROGRAMMABLE LOGIC DEVICES*

In the previous sections, we briefly considered the class of ICs known as programmable logic devices. In Chapter 3, we introduced the concept of describing a circuit's operation using a hardware description language. In this section, we will explore these topics further and become prepared to use the tools of the trade to develop and implement digital systems using PLDs. Of course, it is impossible to understand all the complex details of how a PLD works before grasping the fundamentals of digital circuits. As we examine new fundamental concepts, we will expand our knowledge of the PLDs and the programming methods. The material is presented in such a way that anyone who is not interested in PLDs can easily skip over these sections without loss of continuity in the coverage of the basic principles.

Let's review the process we covered earlier of designing combinational digital circuits. The input devices are identified and assigned an algebraic name like *A*, *B*, *C*, or *LOAD*, *SHIFT*, *CLOCK*. Likewise, output devices are given names like X, Z, or *CLOCK_OUT*, *SHIFT_OUT*. Then a truth table is constructed that lists all the possible input combinations and identifies the required state of the outputs under each input condition. The truth table is one way of describing how the circuit is to operate. Another way to describe the circuit's operation is the Boolean expression. From this point the designer must find the simplest algebraic relationship and select digital ICs that can be wired together to implement the circuit. You have probably experienced that these last steps are the most tedious, time consuming, and prone to errors.

Programmable logic devices allow most of these tedious steps to be automated by a computer and PLD *development software*. Using programmable logic improves the efficiency of the design and development process. Consequently, most modern digital systems are implemented in this way. The job of the circuit designer is to identify inputs and outputs, specify the logical relationship in the most convenient manner, and select a programmable device that is capable of implementing the circuit at the lowest cost. The concept behind programmable logic devices is simple: put lots of logic gates in a single IC and control the interconnection of these gates electronically.

PLD Hardware

Recall from Chapter 3 that many digital circuits today are implemented using programmable logic devices (PLDs). These devices are configured electronically and their internal circuits are "wired" together electronically to form a logic circuit. This programmable wiring can be thought of as thousands of connections that are either connected (1) or not connected (0). It is very tedious to try to configure these devices by manually placing 1s and 0s in a grid, so the next logical question is, "How do we control the interconnection of gates in a PLD electronically?"

---

*All sections covering PLDs may be skipped without loss of continuity in the balance of Chapters 1–12.

A common method of connecting one of many signals entering a network to one of many signal lines exiting the network is a switching matrix. Refer back to Figure 3-44, where this concept was introduced. A matrix is simply a grid of conductors (wires) arranged in rows and columns. Input signals are connected to the columns of the matrix, and the outputs are connected to the rows of the matrix. At each intersection of a row and a column is a switch that can electrically connect that row to that column. The switches that connect rows to columns can be mechanical switches, fusible links, electromagnetic switches (relays), or transistors. This is the general structure used in many applications and will be explored further when we study memory devices in Chapter 12.

PLDs also use a switch matrix that is often referred to as a programmable array. By deciding which intersections are connected and which ones are not, we can "program" the way the inputs are connected to the outputs of the array. In Figure 4-42, a programmable array is used to select the inputs for each AND gate. Notice that in this simple matrix, we can produce any logical product combination of variables A, B at any of the AND gate outputs. A matrix or programmable array such as the one shown in the figure can also be used to connect the AND outputs to OR gates. The details of various PLD architectures will be covered thoroughly in Chapter 13.

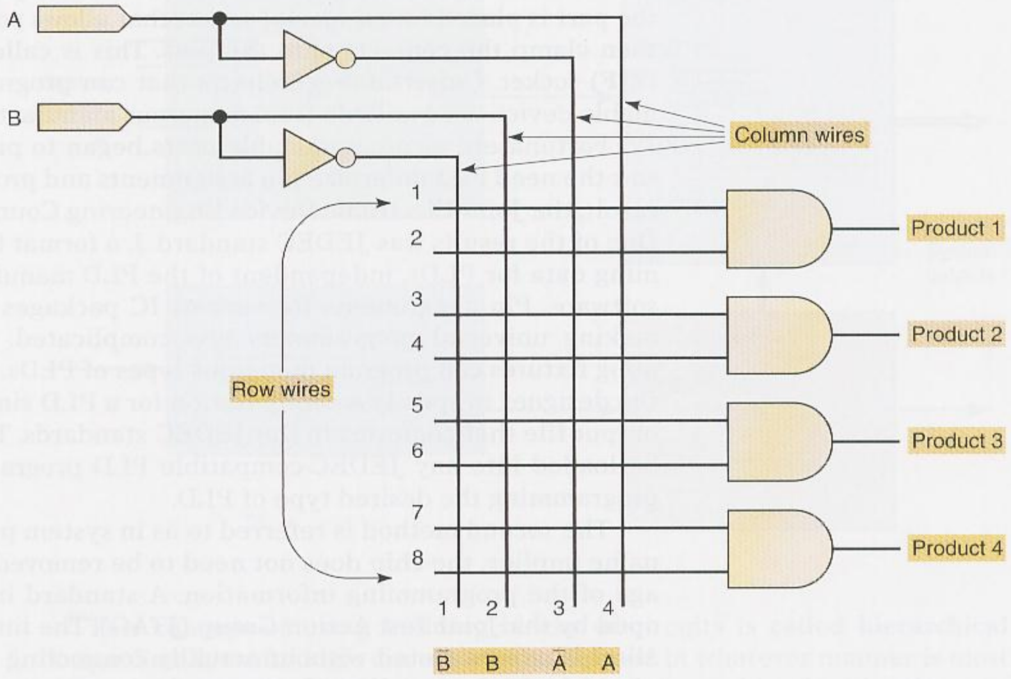
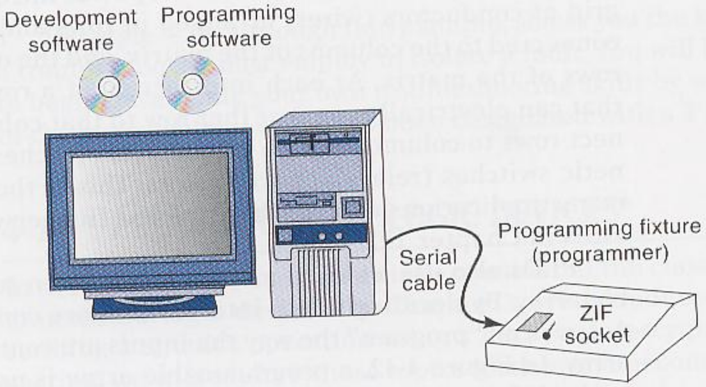


FIGURE 4-42 A programmable array for selecting inputs as product terms.

Programming a PLD

There are two ways to "program" a PLD IC. Programming means making the actual connections in the array. In other words, it means determining which of those connections are supposed to be open (0) and which are supposed to be closed (1). The first method involves removing the PLD IC chip from its circuit board. The chip is then placed in a special fixture called a **programmer**, shown in Figure 4-43. Most modern programmers are connected to a personal computer that is running software containing libraries of information about the many types of programmable devices available.

FIGURE 4-43 A PLD development system.



The programming software is invoked (called up and executed) on the PC to establish communication with the programmer. This software allows the user to set up the programmer for the type of device that is to be programmed, check if the device is blank, read the state of any programmable connection in the device, and provide instructions for the user to program a chip. Ultimately, the part is placed into a special socket that allows you to drop the chip in and then clamp the contacts onto the pins. This is called a **zero insertion force (ZIF)** socket. *Universal programmers* that can program any type of programmable device are available from numerous manufacturers.

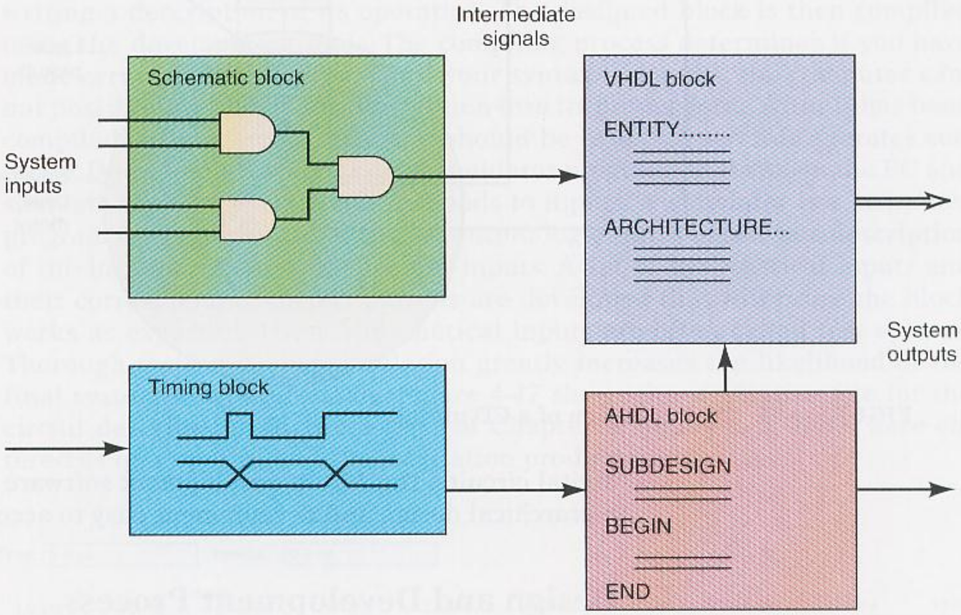
Fortunately, as programmable parts began to proliferate, manufacturers saw the need to standardize pin assignments and programming methods. As a result, the Joint Electronic Device Engineering Council **(JEDEC)** was formed. One of the results was JEDEC standard 3, a format for transferring programming data for PLDs, independent of the PLD manufacturer or programming software. Pin assignments for various IC packages were also standardized, making universal programmers less complicated. Consequently, programming fixtures can program numerous types of PLDs. The software that allows the designer to specify a configuration for a PLD simply needs to produce an output file that conforms to the JEDEC standards. Then this JEDEC file can be loaded into any JEDEC-compatible PLD programmer that is capable of programming the desired type of PLD.

The second method is referred to as in system programming (ISP). As its name implies, the chip does not need to be removed from its circuit for storage of the programming information. A standard interface has been developed by the Joint Test Action Group **(JTAG)**. The interface was developed to allow ICs to be tested without actually connecting test equipment to every pin of the IC. It also allows for internal programming. Four pins on the IC are used as a portal to store data and retrieve information about the inner condition of the IC. Many ICs, including PLDs and microcontrollers, are manufactured today to include the JTAG interface. An interface cable connects the four JTAG pins on the IC to an output port (like the printer port) of a personal computer. Software running on the PC establishes contact with the IC and loads the information in the proper format.

Development Software

We have examined several methods of describing logic circuits now, including schematic capture, logic equations, truth tables, and HDL. We also described the fundamental methods of storing 1s and 0s into a PLD IC to connect the logic circuits in the desired way. The biggest challenge in getting a PLD programmed is converting from any form of description into the array of 1s and 0s. Fortunately, this task is accomplished quite easily by a computer running the development software. The development software that we will be referring to and using for examples is produced by Altera. This software allows the designer to enter a circuit description in any one of the many ways we have been discussing: graphic design files (schematics), AHDL, and VHDL. It also allows the use of another HDL, called Verilog, and the option of describing the circuit with timing diagrams. Circuit blocks described by any of these methods can also be "connected" together to implement a much larger digital system, as shown in Figure 4-44. Any logic diagram found in this text can be redrawn using the schematic entry tools in the Altera software to create a graphic design file. We will not focus on graphic design entry in this text because it is quite straightforward to pick up these skills in the laboratory. We will focus our examples on the methods that allow us to use HDL as an alternate means of describing a circuit. For more information on the Altera software, see the accompanying CD as well as user manuals from the Altera web site (http://www.altera.com).

FIGURE 4-44 Combining blocks developed using different description methods.



This concept of using building blocks of circuits is called **hierarchical design**. Small, useful logic circuits can be defined in whatever manner is most convenient (graphic, HDL, timing, etc.) and then combined with other circuits to form a large section of a project. Sections can be combined and connected with other sections to form the whole system. Figure 4-45 shows the hierarchical structure of a CD player using a block diagram. The outer box encloses the entire system. The dashed lines identify each major subsection, and each subsection contains individual circuits. Although it is not shown in this diagram, each circuit may be made up of smaller building blocks of common

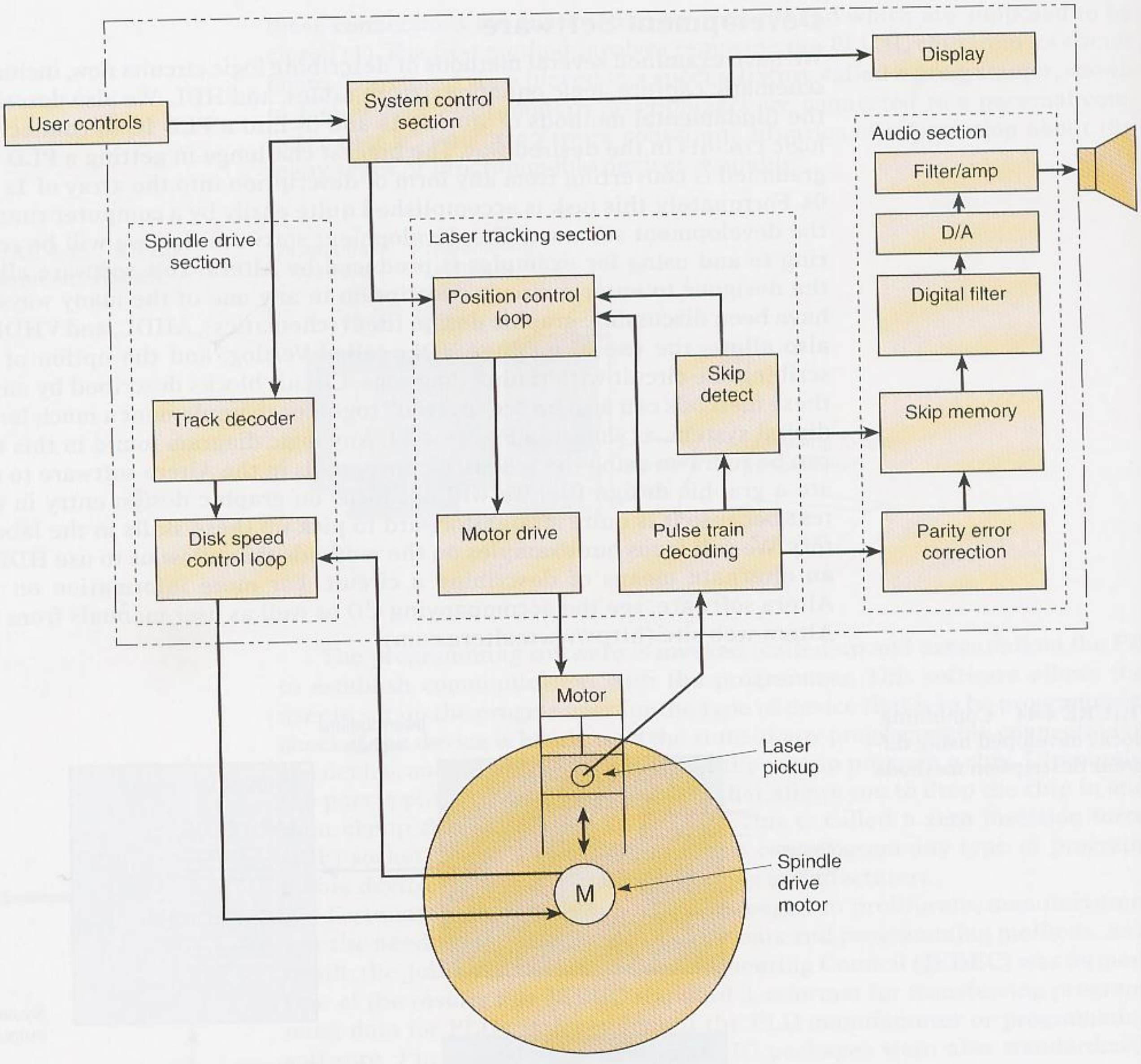

Display
User controls
System control section
Audio section
Filter/amp
D/A
Digital filter
Spindle drive section
Laser tracking section
Position control loop
Skip detect
Track decoder
Skip memory
Disk speed control loop
Motor drive
Pulse train decoding
Parity error correction
Motor
Laser pickup
M
Spindle drive motor

FIGURE 4-45 Block diagram of a CD player.

digital circuits. The Altera development software makes this type of modular, hierarchical design and development easy to accomplish.

Design and Development Process

Another way you might see the hierarchy of a system like the CD player just described is shown in Figure 4-46. The top level represents the entire system. It is made up of three subsections, each of which in turn is made up of the smaller circuits shown. Notice that this diagram does not show how the signals flow throughout the system but clearly identifies the various levels of the hierarchical structure of the project.

This type of diagram has led to the name for one of the most common methods of design: **top-down**. With this design approach, you start with the overall description of the entire system, such as the top box in Figure 4-46. Then you define several subsections that will make up the system. The subsections are further refined into individual circuits connected together.

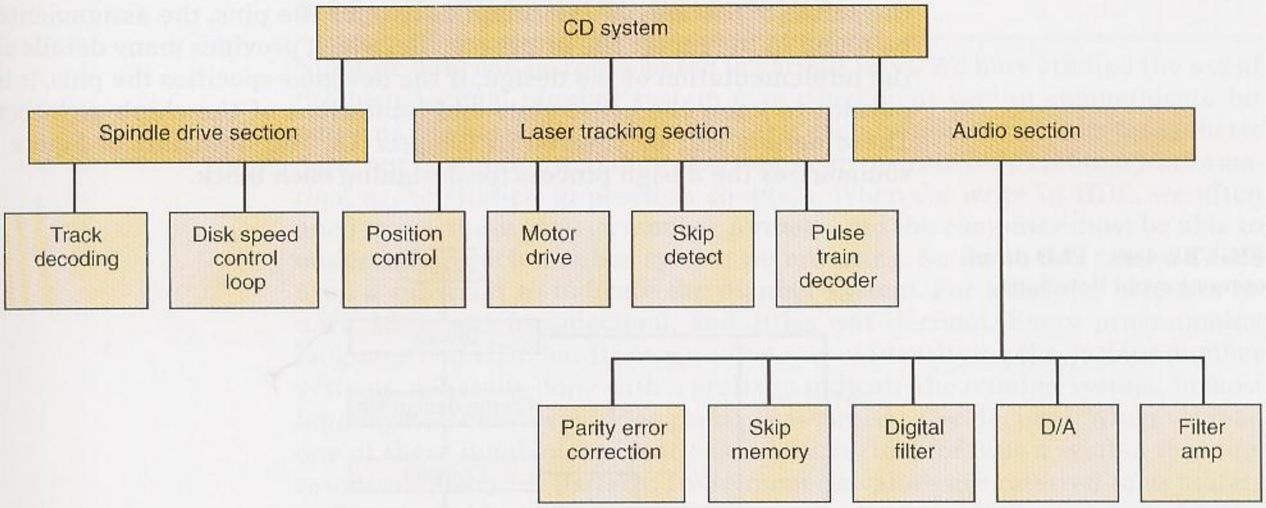

CD system
Spindle drive section
Laser tracking section
Audio section
Track decoding
Disk speed control loop
Position control
Motor drive
Skip detect
Pulse train decoder
Parity error correction
Skip memory
Digital filter
D/A
Filter amp

FIGURE 4-46 An organizational hierarchy chart.

Every one of these hierarchy levels has defined inputs, outputs, and behavior. Each can be tested individually before it is connected to the others.

After defining the blocks from the top down, the system is built from the bottom up. Each block in this system design has a design file that describes it. The lowest level blocks must be designed by opening a design file and writing a description of its operation. The designed block is then compiled using the development tools. The compiling process determines if you have made errors in your syntax. Until your syntax is correct, the computer cannot possibly translate your description into its proper form. After it has been compiled with no syntax errors, it should be tested to see if it operates correctly. Development systems offer simulator programs that run on the PC and simulate the way your circuit responds to inputs. A simulator is a computer program that calculates the correct output logic states based on a description of the logic circuit and the current inputs. A set of hypothetical inputs and their corresponding correct outputs are developed that will prove the block works as expected. These hypothetical inputs are often called **test vectors**. Thorough testing during simulation greatly increases the likelihood of the final system working reliably. Figure 4-47 shows the simulation file for the circuit described in Figure 3-13(a) of Chapter 3. Inputs *a*, *b*, and *c* were entered as test vectors, and the simulation produced output *y*.

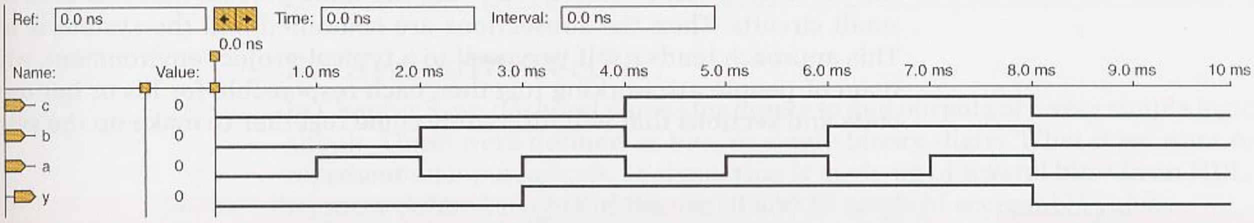

Ref: 0.0 ns
Time: 0.0 ns
Interval: 0.0 ns
0.0 ns
Name:
Value:
1.0 ms
2.0 ms
3.0 ms
4.0 ms
5.0 ms
6.0 ms
7.0 ms
8.0 ms
9.0 ms
10 ms
c 0
b 0
a 0
y 0

FIGURE 4-47 A timing simulation of a circuit described in HDL.

When the designer is satisfied that the design works, the design can be verified by actually programming a chip and testing. For a complex PLD, the designer can either let the development system assign pins and then lay out the final circuit board accordingly, or specify the pins for each signal using the software features. If the compiler assigns the pins, the assignments can be found in the report file or pin-out file, which provides many details about the implementation of the design. If the designer specifies the pins, it is important to know the constraints and limitations of the chip's architecture. These details will be covered in Chapter 13. The flowchart of Figure 4-48 summarizes the design process for designing each block.

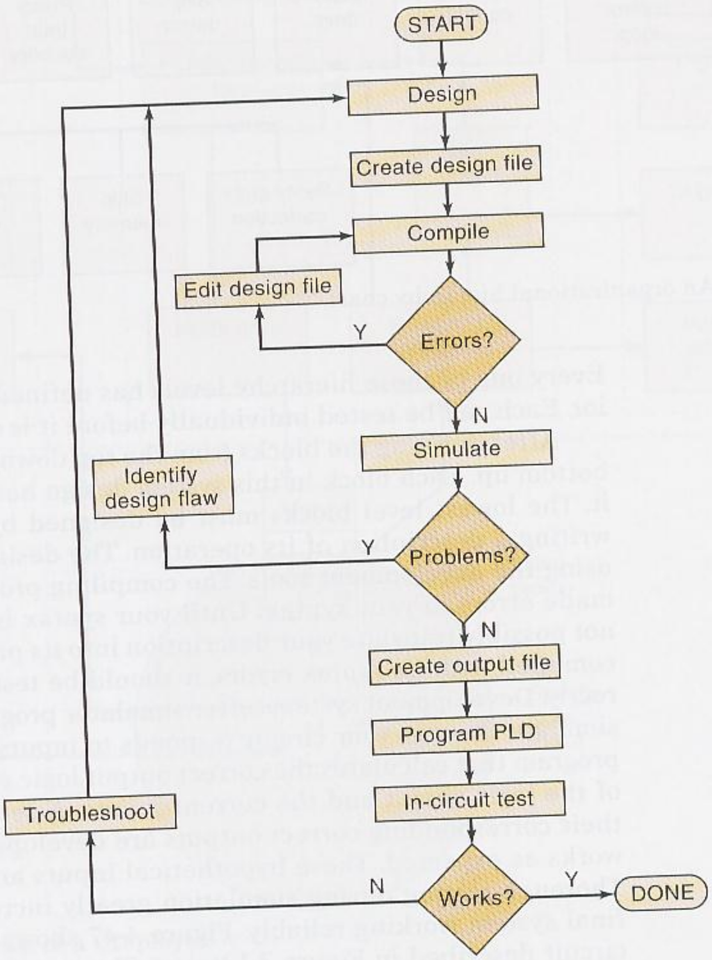

START
Design
Create design file
Compile
Edit design file
Y
Errors?
N
Simulate
Identify design flaw
Y
Problems?
N
Create output file
Program PLD
In-circuit test
Troubleshoot
N
Works?
Y
DONE

FIGURE 4-48 PLD development cycle flowchart.

After each circuit in a subsection has been tested, all can be combined and the subsection can be tested following the same process that was used for the small circuits. Then the subsections are combined and the system is tested. This approach lends itself very well to a typical project environment, where a team of people are working together, each responsible for his or her own circuits and sections that will ultimately come together to make up the system.

REVIEW QUESTIONS

1. What is actually being "programmed" in a PLD?
2. What bits (column, row) in Figure 4-42 must be connected to make Product 1 = AB?
3. What bits (column, row) in Figure 4-42 must be connected to make Product 3 = $A\bar{B}$?

4-15 REPRESENTING DATA IN HDL

Numeric data can be represented in various ways. We have studied the use of the hexadecimal number system as a convenient way to communicate bit patterns. We naturally prefer to use the decimal number system for numeric data, but computers and digital systems can operate only on binary information, as we studied in previous chapters. When we write in HDL, we often need to use these various number formats, and the computer must be able to understand which number system we are using. So far in this text, we have used a subscript to indicate the number system. For example, 101_2 was binary, 101_{16} was hexadecimal, and 101_{10} was decimal. Every programming language and HDL has its own unique way of identifying the various number systems, generally done with a prefix to indicate the number system. In most languages, a number with no prefix is assumed to be decimal. When we read one of these number designations, we must think of it as a symbol that represents a binary bit pattern. These numeric values are referred to as scalars or **literals**. Table 4-8 summarizes the methods of specifying values in binary, hex, and decimal for AHDL and VHDL.

TABLE 4-8 Designating number systems in HDL.

Number System	AHDL	VHDL	Bit Pattern	Decimal Equivalent
Binary	B"101"	B"101"	101	5
Hexadecimal	H"101"	X"101"	100000001	257
Decimal	101	101	1100101	101

EXAMPLE 4-29

Express the following bit pattern's numeric value in binary, hex, and decimal using AHDL and VHDL notation:

11001

Solution

Binary is designated the same in both AHDL and VHDL: **B "11001"**.
Converting the binary to hex, we have 19_{16}.
In AHDL: **H "19"**
In VHDL: **X "19"**
Converting the binary to decimal, we have 25_{10}.
Decimal is designated the same in both AHDL and VHDL: 25.

Bit Arrays/Bit Vectors

In Chapter 3, we declared names for inputs to and outputs of a very simple logic circuit. These were defined as bits, or single binary digits. What if we want to represent an input, output, or signal that is made up of several bits? In an HDL, we must define the type of the signal and its range of acceptable values.

To understand the concepts used in HDLs, let's first consider some conventions for describing bits of binary words in common digital systems. Suppose we have an eight-bit number representing the current temperature, and the number is coming into our digital system through an input port that we have named P1, as shown in Figure 4-49. We can refer to the individual bits of this port as P1 bit 0 for the least significant bit, on up to P1 bit 7 for the most significant bit.

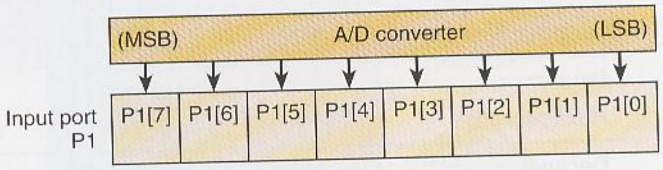
We can also describe this port by saying that it is named P1, with bits numbered 7 down to 0. The terms **bit array** and **bit vector** are often used to describe this type of data structure. It simply means that the overall data structure (eight-bit port) has a name (P1) and that each individual element (bit) has a unique **index** number (0–7) to describe that bit's position (and possibly its numeric weight) in the overall structure. The HDLs and computer programming languages take advantage of this notation. For example, the third bit from the right is designated as P1[2], and it can be connected to another signal bit by using an assignment operator.

EXAMPLE 4-30

Assume there is an eight-bit array named P1, as shown in Figure 4-49, and another four-bit array is named P5.

(a) Write the bit designation for the most significant bit of P1.
(b) Write the bit designation for the least significant bit of P5.
(c) Write an expression that causes the least significant bit of P5 to drive the most significant bit of P1.

FIGURE 4-49 Bit array notation.



Solution

(a) The name of the port is P1 and the most significant bit is bit 7. The proper designation for P1 bit 7 is P1[7].
(b) The name of the port is P5 and the least significant bit is bit 0. The proper designation for P5 bit 0 is P5[0].
(c) The driving signal is placed on the right side of the assignment operator, and the driven signal is placed on the left: P1[7] = P5[0];.

AHDL

AHDL BIT ARRAY DECLARATIONS

In AHDL, port *p1* of Figure 4-49 is defined as an eight-bit input port, and the value on this port can be referred to using any number system, such as hex, binary, decimal, etc. The syntax for AHDL uses a name for the bit vector followed by the range of index designations, which are enclosed in square brackets. This declaration is included in the SUBDESIGN section. For example, to declare an eight-bit input port called *p1*, you would write

```
p1 [7..0] :INPUT; --define an 8-bit input port
```

EXAMPLE 4-31

Declare a four-bit input named *keypad* using AHDL.

Solution

```
keypad [3..0] :INPUT;
```

Intermediate variables can also be declared as an array of bits. As with single bits, they are declared just after the I/O declarations in SUBDESIGN. As an example, the eight-bit temperature port *p1* can be assigned (connected) to a node named *temp*, as follows:

```
VARIABLE temp [7..0] :NODE;
BEGIN
   temp[] = p1[];
END;
```

Notice that the input port *p1* has the data applied to it, and it is driving the signal wires named *temp*. Think of the term on the right of the equals sign as the source of the data and the term on the left as the destination. The empty brackets [] mean that each of the correspanding bits in the two arrays are being connected. Individual bits can also be "connected" by specifying the bits inside the brackets. For example, to connect only the least significant bit of p1 to the LSB of temp, the statement would be temp[0] = p1[0];.

VHDL BIT VECTOR DECLARATIONS

In VHDL, port *p1* of Figure 4-49 is defined as an eight-bit input port, and the value on this port can be referred to using only binary literals. The syntax for VHDL uses a name for the bit vector followed by the mode (:IN), the type (**BIT_VECTOR**), and the range of index designations, which are enclosed in parentheses. This declaration is included in the ENTITY section. For example, to declare an eight-bit input port called *p1*, you would write

```
PORT (p1 :IN BIT_VECTOR (7 DOWNTO 0);
```

EXAMPLE 4-32

Declare a four-bit input named *keypad* using VHDL.

Solution

```
PORT(keypad :IN BIT_VECTOR (3 DOWNTO 0);
```

Intermediate signals can also be declared as an array of bits. As with single bits, they are declared just inside the ARCHITECTURE definition. As an example, the eight-bit temperature on port *p1* can be assigned (connected) to a signal named *temp*, as follows:

```
SIGNAL    temp :BIT_VECTOR (7 DOWNTO 0);
BEGIN
    temp <= p1;
END;
```

Notice that the input port *p1* has the data applied to it, and it is driving the signal wires named *temp*. No elements in the bit vector are specified, which means that all the bits are being connected. Individual bits can also be "connected" using signal assignments and by specifying the bit numbers inside parentheses. For example, to connect only the least significant bit of *p1* to the LSB of *temp*, the statement would be temp(0) <= p1(0);.

VHDL is very particular regarding the definitions of each type of the data. The type "bit_vector" describes an array of individual bits. This is interpreted differently than an eight-bit binary number (called a scaler quantity), which has the type **integer**. Unfortunately, VHDL does not allow us to assign an integer value to a BIT_VECTOR signal directly. Data can be represented by any of the types shown in Table 4-9, but data assignments and other operations must be done between objects of the same type. For example, the compiler will not allow you to take a number from a keypad declared as an integer and connect it to four LEDs that are declared as BIT_VECTOR outputs. Notice in Table 4-9, under Possible Values, that individual BIT and STD_LOGIC data **objects** (e.g., signals, variables, inputs, and outputs) are designated by single quotes, whereas values assigned to BIT_VECTOR and STD_LOGIC_VECTOR types are strings of valid bit values enclosed in double quotes.

TABLE 4-9 Common VHDL data types.

Data Type	Sample Declaration	Possible Values	Use
BIT	y :OUT BIT;	'0' '1'	y <= '0';
STD_LOGIC	driver :STD_LOGIC	'0' '1' 'z' 'x' '-'	driver <= 'z';
BIT_VECTOR	bcd_data :BIT_VECTOR (3 DOWNTO 0);	"0101" "1001" "0000"	digit <= bcd_data;
STD_LOGIC_VECTOR	dbus :STD_LOGIC_VECTOR (3 DOWNTO 0);	"0Z1X"	IF rd = '0' THEN dbus <= "zzzz";
INTEGER	SIGNAL z:INTEGER RANGE −32 TO 31;	−32..−2,−1,0,1,2 . . . 31	IF z > 5 THEN . . .

VHDL also offers some standardized data types that are necessary when using logic functions that are contained in the **libraries**. As you might have guessed, libraries are simply collections of little pieces of VHDL code that can be used in your hardware descriptions without reinventing the wheel. These libraries offer convenient functions, called **macrofunctions**, like many of the standard TTL devices that are described throughout this text. Rather than writing a new description of a familiar TTL device, we can simply pull its macrofunction out of the library and use it in our system. Of course, you need to get signals into and out of these macrofunctions, and the types of the signals in your code must match the types in the functions (which someone else wrote). This means that everyone must use the same standard data types.

When VHDL was standardized through the IEEE, many data types were created at the same time. The two that we will use in this text are **STD_LOGIC**, which is equivalent to BIT type, and **STD_LOGIC_VECTOR**, which is equivalent to BIT_VECTOR. As you recall, BIT type can have values of only '0' and '1'. The standard logic types are defined in the IEEE library and have a broader range of possible values than their built-in counterparts. The possible values for a STD_LOGIC type or for any element in a STD_LOGIC_VECTOR are given in Table 4-10. The names of these categories will make much more sense after we study the characteristics of logic circuits in Chapter 8. For now, we will show examples using values of only '1' and '0'.

TABLE 4-10 STD_LOGIC values.

'1'	Logic 1 (just like BIT type)
'0'	Logic 0 (just like BIT type)
'z'	High impedance*
'-'	don't care (just like you used in your K maps)
'U'	Uninitialized
'X'	Unknown
'W'	Weak unknown
'L'	Weak '0'
'H'	Weak '1'

*We will study tristate logic in Chapter 8.

REVIEW QUESTIONS

1. How would you declare a six-bit input array named push_buttons in (a) AHDL or (b) VHDL?
2. What statement would you use to take the MSB from the array in question 1 and put it on a single-bit output port named z? Use (a) AHDL or (b) VHDL.
3. In VHDL, what is the IEEE standard type that is equivalent to the BIT type?
4. In VHDL, what is the IEEE standard type that is equivalent to the BIT_VECTOR type?

4-16 TRUTH TABLES USING HDL

We have learned that a truth table is another way of expressing the operation of a circuit block. It relates the output of the circuit to every possible combination of its inputs. As we saw in Section 4-4, a truth table is the starting point for a designer to define how the circuit should operate. Then a Boolean expression is derived from the truth table and simplified using K maps or Boolean algebra. Finally the circuit is implemented from the final Boolean equation. Wouldn't it be great if we could go from the truth table directly to the final circuit without all those steps? We can do exactly that by entering the truth table using HDL.

TRUTH TABLES USING AHDL

The code in Figure 4-50 uses AHDL to implement a circuit and uses a truth table to describe its operation. The truth table for this design was presented in Example 4-7. The key point of this example is the use of the TABLE keyword in AHDL. It allows the designer to specify the operation of the circuit just like you would fill out a truth table. On the first line after TABLE, the input variables (a,b,c) are listed exactly like you would create a column heading on a truth table. By including the three binary variables in parentheses, we tell the compiler that we want to use these three bits as a group and to refer to them as a three-bit binary number or bit pattern. The specific values for this bit pattern are listed below the group and are referred to as binary literals. The special operator (=>) is used in truth tables to separate the inputs from the output (y).

FIGURE 4-50 AHDL design file for Figure 4-7

```
%     Figure 4-7 in AHDL
      Digital Systems 10th ed
      Neal Widmer
      MAY 23, 2005                                  %
SUBDESIGN FIG4_50
(
      a,b,c :INPUT;                     --define inputs to block
      y      :OUTPUT;                   --define block output

)
BEGIN
      TABLE
            (a,b,c)                     =>    y;    --column headings
            (0,0,0)                     =>    0;
            (0,0,1)                     =>    0;
            (0,1,0)                     =>    0;
            (0,1,1)                     =>    1;
            (1,0,0)                     =>    0;
            (1,0,1)                     =>    1;
            (1,1,0)                     =>    1;
            (1,1,1)                     =>    1;
      END TABLE;
END;
```

The TABLE in Figure 4-50 is intended to show the relationship between the HDL code and a truth table. A more common way of representing the input data heading is to use a variable bit array to represent the value on *a*, *b*, *c*. This method involves a declaration of the bit array on the line before BEGIN, such as:

```
VARIABLE in_bits[2..0]     :NODE;
```

Just before the TABLE keyword, the input bits can be assigned to the array, *inbits[ ]*:

```
in_bits[] = (a,b,c);
```

Grouping three independent bits in order like this is referred to as **concatenating**, and it is often done to connect individual bits to a bit array. The table heading on the input bit sets can be represented by *in_bits[ ]*, in this case. Note that as we list the possible combinations of the inputs, we have several options. We can make up a group of 1s and 0s in parentheses, as shown in Figure 4-50, or we can represent the same bit pattern using the equivalent binary, hex, or decimal number. It is up to the designer to decide which format is most appropriate depending on what the input variables represent.

TRUTH TABLES USING VHDL: SELECTED SIGNAL ASSIGNMENT

The code in Figure 4-51 uses VHDL to implement a circuit using a selected **signal assignment** to describe its operation. It allows the designer to specify the operation of the circuit, just like you would fill out a truth table. The truth table for this design was presented in Example 4-7. The primary point of this example is the use of the WITH signal_name SELECT statement in VHDL. A secondary point presented here shows how to put the data into a

```
--     Figure 4-7 in VHDL
--     Digital Systems 10th ed
--     Neal Widmer
--     MAY 23, 2005
ENTITY fig4_51 IS
PORT(
       a,b,c :IN BIT;               --declare individual input bits
       y     :OUT BIT);
END fig4_51;

ARCHITECTURE truth OF fig4_51 IS
       SIGNAL in_bits :BIT_VECTOR(2 DOWNTO 0);
       BEGIN
       in_bits <= a & b & c;        --concatenate input bits into bit_vector
             WITH in_bits SELECT
             y     <=      '0'  WHEN "000",    --Truth Table
                           '0'  WHEN "001",
                           '0'  WHEN "010",
                           '1'  WHEN "011",
                           '0'  WHEN "100",
                           '1'  WHEN "101",
                           '1'  WHEN "110",
                           '1'  WHEN "111";
END truth;
```

FIGURE 4-51 VHDL design file for Figure 4-7.

format that can be used conveniently with the selected signal assignment. Notice that the inputs are defined in the ENTITY declaration as three independent bits *a*, *b*, and *c*. Nothing in this declaration makes one of these more significant than another. The order in which they are listed does not matter. We want to compare the current value of these bits with each of the possible combinations that could be present. If we drew out a truth table, we would decide which bit to place on the left (MSB) and which to place on the right (LSB). This is accomplished in VHDL by **concatenating** (connecting in order) the bit variables to form a bit vector. The concatenation operator is "&". A signal is declared as a BIT_VECTOR to receive the ordered set of input bits and is used to compare the input's value with the string literals contained in quotes. The output (*y*) is assigned (<=) a bit value ('0' or '1') WHEN *in_bits* contains the value listed in double quotes.

VHDL is very strict in the way it allows us to assign and compare objects such as signals, variables, constants, and literals. The output *y* is a BIT, and so it must be assigned a value of '0' or '1'. The SIGNAL *in_bits* is a three-bit BIT_VECTOR, so it must be compared with a three-bit string literal value. VHDL will not allow *in_bits* (a BIT_VECTOR) to be compared with a hex number like X "5" or a decimal number like 3. These scalar quantities would be valid for assignment or comparison with integers.

EXAMPLE 4-33

Declare three signals in VHDL that are single bits named *too_hot, too_cold,* and *just_right.* Combine (concatenate) these three bits into a three-bit signal called *temp_status,* with hot on the left and cold on the right.

Solution

1. Declare signals first in Architecture.

```
SIGNAL too_hot, too_cold, just_right :BIT;
SIGNAL temp_status :BIT_VECTOR (2 DOWNTO 0);
```

2. Write concurrent assignment statements between BEGIN and END.

```
temp_status <= too_hot & just_right & too_cold;
```

REVIEW QUESTIONS

1. How would you concatenate three bits x, y, and z into a three-bit array named *omega*? Use AHDL or VHDL.
2. How are truth tables implemented in AHDL?
3. How are truth tables implemented in VHDL?

4-17 DECISION CONTROL STRUCTURES IN HDL

In this section, we will examine methods that allow us to tell the digital system how to make "logical" decisions in much the same way that we make decisions every day. In Chapter 3, we learned that concurrent assignment statements are evaluated such that the order in which they are written has no effect on the circuit being described. When using **decision control structures,** the order in which we ask the questions does matter. To summarize this concept in the terms used in HDL documentation, statements that can be written in any sequence are called **concurrent**, and statements that are evaluated in the sequence in which they are written are called **sequential**. The sequence of sequential statements affects the circuit's operation.

The examples we have considered so far involve several individual bits. Many digital systems require inputs that represent a numeric value. Refer again to Example 4-8, in which the purpose of the logic circuit is to monitor the battery voltage measured by an A/D converter. The digital value is represented by a four-bit number coming from the A/D into the logic circuit. These inputs are not independent binary variables but rather four binary digits of a number representing battery voltage. We need to give the data the correct type that will allow us to use it as a number.

IF/ELSE

Truth tables are great for listing all the possible combinations of independent variables, but there are better ways to handle numeric data. As an example, when a person leaves for school or work in the morning, she must make a logical decision about wearing a coat. Let's assume she decides this issue based only on the current temperature. How many of us would reason as follows?

I will wear a coat if the temperature is 0.

I will wear a coat if the temperature is 1.

I will wear a coat if the temperature is 2. . . .

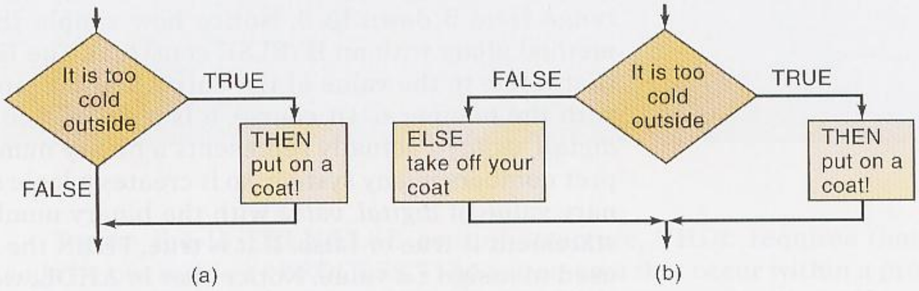
I will wear a coat if the temperature is 55.

I will *not* wear a coat if the temperature is 56.
I will *not* wear a coat if the temperature is 57.
I will *not* wear a coat if the temperature is 58. . . .
I will *not* wear a coat if the temperature is 99.

This method is similar to the truth table approach of describing the decision. For every possible input, she decides what the output should be. Of course, what she would really do is decide as follows:

I will wear a coat if the temperature is less than 56 degrees.
Otherwise, I will *not* wear a coat.

An HDL gives us the power to describe logic circuits using this type of reasoning. First, we must describe the inputs as a *number within a given range,* and then we can write statements that decide what to do to the outputs based on the *value* of the incoming number. In most computer programming languages, as well as HDLs, these types of decisions are made using an IF/THEN/ELSE control structure. Whenever the decision is between doing something and doing nothing, an **IF/THEN** construct is used. The keyword IF is followed by a statement that is true or false. IF it is true, THEN do whatever is specified. In the event that the statement is false, no action is taken. Figure 4-52(a) shows graphically how this decision works. The diamond shape represents the decision being made by evaluating the statement contained within the diamond. Every decision has two possible outcomes: true or false. In this example, if the statement is false, no action is taken.



(a) (b)

FIGURE 4-52 Logical flow of (a) IF/THEN and (b) IF/THEN/ELSE constructs.

In some cases it is not enough only to decide to act or not to act, but rather we must choose between two different actions. For example, in our analogy about the decision to wear a coat, if the person already has her coat on when making this decision, she will not be taking it off. The use of IF/THEN logic assumes that she is not wearing her coat initially.

When decisions demand two possible actions, the IF/THEN/**ELSE** control structure is used, as shown in Figure 4-52(b). Here again, the statement is evaluated as true or false. The difference is that, when the statement is false, a different action is performed. One of the two actions must occur with this construct. We can state it verbally as, "IF the statement is true, THEN do this. ELSE do that." In our coat analogy, this control structure would work, regardless of whether the person's coat was on or off initially.

Example 4-8 gave a simple example of a logic circuit that has as its input a numeric value representing battery voltage from an A/D converter. The inputs A, B, C, D are actually binary digits in a four-bit number, with A being the MSB and D being the LSB. Figure 4-53 shows the same circuit with the

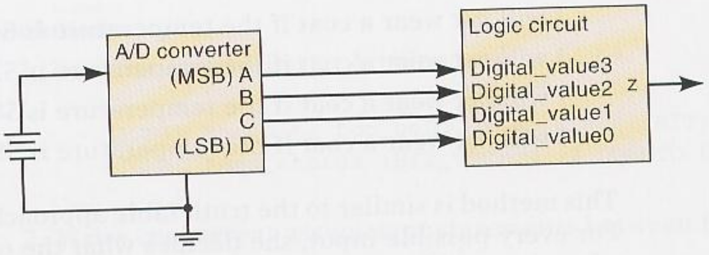


FIGURE 4-53 Logic circuit similar to Example 4-8.

inputs labeled as a four-bit number called *digital_value.* The relationship between bits is as follows:

A	*digital_value[3]*	digital value bit 3 (MSB)
B	*digital_value[2]*	digital value bit 2
C	*digital_value[1]*	digital value bit 1
D	*digital_value[0]*	digital value bit 0 (LSB)

The input can be treated as a decimal number between 0 and 15 if we specify the correct type of the input variable.

IF/THEN/ELSE USING AHDL

In AHDL, the inputs can be specified as a binary number made up of multiple bits by assigning a variable name followed by a list of the bit positions, as shown in Figure 4-54. The name is *digital_value*, and the bit positions range from 3 down to 0. Notice how simple the code becomes using this method along with an IF/ELSE construct. The IF is followed by a statement that refers to the value of the entire four-bit input variable and compares it with the number 6. Of course, 6 is a decimal form of a scalar quantity and *digital_value[ ]* actually represents a binary number. The compiler can interpret numbers in any system, so it creates a logic circuit that compares the binary value of *digital_value* with the binary number for 6 and decides if this statement is true or false. If it is true, THEN the next statement (z = VCC) is used to assign *z* a value. Notice that in AHDL, we must use VCC for a logic 1 and GND for a logic 0 when assigning a logic level to a single bit. When *digital_value* is 6 or less, it follows the statement after ELSE (z = GND). The END IF; terminates the control structure.

FIGURE 4-54 AHDL version.

```
SUBDESIGN FIG4_54
(
   digital_value[3..0]   :INPUT;  -- define inputs to block
   z                     :OUTPUT; -- define block output
)
BEGIN
   IF digital_value[] > 6 THEN
         z = VCC;                         -- output a 1
   ELSE  z = GND;                         -- output a 0
   END IF;
END;
```

IF/THEN/ELSE USING VHDL

In VHDL, the critical issue is the declaration of the type of inputs. Refer to Figure 4-55. The input is treated as a single variable called *digital_value*. Because its type is declared as INTEGER, the compiler knows to treat it as a number. By specifying a range of 0 to 15, the compiler knows it is a four-bit number. Notice that RANGE does not specify the index number of a bit vector but rather the limits of the numeric value of the integer. Integers are treated differently than bit arrays (BIT_VECTOR) in VHDL. An integer can be compared with other numbers using inequality operators. A BIT_VECTOR cannot be used with inequality operators.

FIGURE 4-55 VHDL version.

```
ENTITY fig4_55 IS
PORT( digital_value :IN INTEGER RANGE 0 TO 15; -- 4-bit input
      z             :OUT BIT);
END fig4_55;

ARCHITECTURE decision OF fig4_55 IS

BEGIN
   PROCESS (digital_value)
      BEGIN
         IF (digital_value > 6) THEN
            z <= '1';
         ELSE
            z <= '0';
      END IF;
END PROCESS ;
END decision;
```

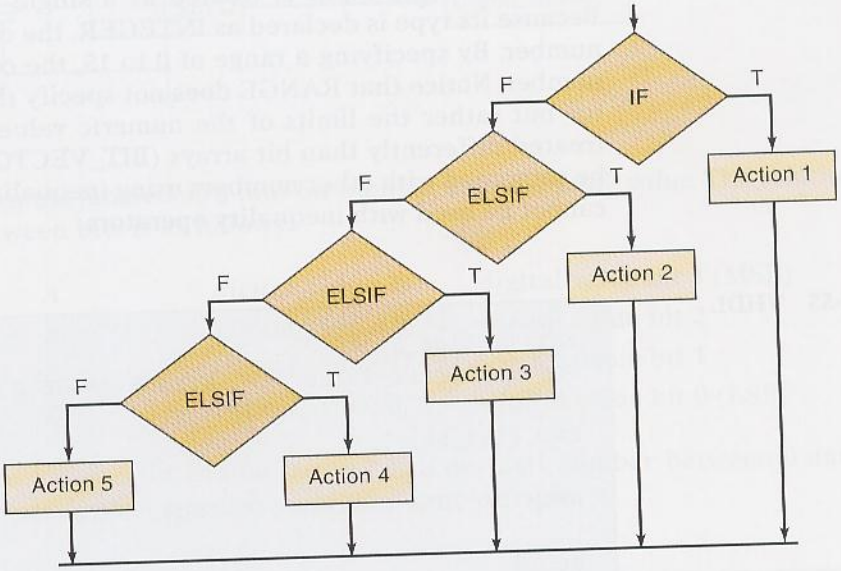
To use the IF/THEN/ELSE control structure, VHDL requires that the code be put inside a "PROCESS." The statements that occur within a process are *sequential*, meaning that the order in which they are written affects the operation of the circuit. The keyword **PROCESS** is followed by a list of variables called a **sensitivity list**, which is a list of variables to which the process code must respond. Whenever *digital_value* changes, it causes the process code to be reevaluated. Even though we know *digital_value* is really a four-bit binary number, the compiler will evaluate it as a number between the equivalent decimal values of 0 and 15. IF the statement in parentheses is true, THEN the next statement is applied (*z* is assigned a value of logic 1). If this statement is not true, the logic follows the ELSE clause and assigns a value of 0 to *z*. The END IF; terminates the control structure, and the END PROCESS; terminates the evaluation of the sequential statements.

ELSIF

We often need to choose among many possible actions, depending on the situation. The IF construct chooses whether to perform a set of actions or not. The IF/ELSE construct selects one out of two possible actions. By combining

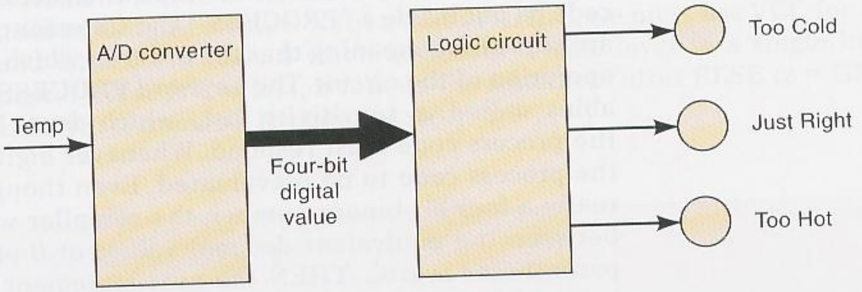
IF and ELSE decisions, we can create a control structure referred to as **ELSIF**, which chooses one of many possible outcomes. The decision structure is shown graphically in Figure 4-56.

FIGURE 4-56 Flowchart for multiple decisions using IF/ELSIF.


IF
T
F
Action 1
ELSIF
T
F
Action 2
ELSIF
T
F
Action 3
ELSIF
T
F
Action 4
Action 5

Notice that as each condition is evaluated, it either performs an action if true or goes on to evaluate the next condition. Each action is associated with one condition, and there is no chance to select more than one action. Note also that the conditions used to decide the appropriate action can be any expression that evaluates as true or false. This fact allows the designer to use the inequality operators to choose an action based on a range of input values. As an example of this application, let's consider the temperature-measuring system that uses an A/D converter, as described in Figure 4-57. Suppose that we want to indicate when the temperature is in a certain range, which we will refer to as Too Cold, Just Right, and Too Hot.

FIGURE 4-57 Temperature range indicator circuit.


A/D converter
Temp
Four-bit digital value
Logic circuit
Too Cold
Just Right
Too Hot

The relationship between the digital values for temperature and the categories is

Digital Values	*Category*
0000–1000	Too Cold
1001–1010	Just Right
1011–1111	Too Hot

We can express the decision-making process for this logic circuit as follows:

> IF the digital value is less than or equal to 8, THEN light only the Too Cold indicator.
>
> ELSE IF the digital value is greater than 8 AND less than 11, THEN light only the Just Right indicator.
>
> ELSE light only the Too Hot indicator.

ELSIF USING AHDL

The AHDL code in Figure 4-58 defines the inputs as a four-bit binary number. The outputs are three individual bits that drive the three range indicators. This example uses an intermediate variable (*status*) that allows us to assign a bit pattern representing the three conditions of *too_cold, just_right,* and *too_hot*. The sequential section of the code uses the IF, ELSIF, ELSE to identify the range in which the temperature lies and assigns the correct bit pattern to status. In the last statement, the bits of *status* are connected to the actual output port bits. These bits have been ordered in a group that relates to the bit patterns assigned to *status[ ]*. This could also have been written as three concurrent statements: too_cold = status[2]; just_right = status[1]; too_hot = status[0];.

```
SUBDESIGN fig4_58
(
   digital_value[3..0]            :INPUT;  --define inputs to block
   too_cold, just_right, too_hot :OUTPUT;--define outputs
)
VARIABLE
status[2..0]     :NODE;--holds state of too_cold, just_right, too_hot
BEGIN
   IF       digital_value[] <= 8 THEN status[] = b"100";
   ELSIF    digital_value[] > 8 AND digital_value[] < 11 THEN
         status[] = b"010";
   ELSE  status[] = b"001";
   END IF;
   (too_cold, just_right, too_hot) = status[]; -- update output bits
END;
```

FIGURE 4-58 Temperature range example in AHDL using ELSIF.

ELSIF USING VHDL

The VHDL code in Figure 4-59 defines the inputs as a four-bit integer. The outputs are three individual bits that drive the three range indicators. This example uses an intermediate signal (*status*) that allows us to assign a bit pattern representing all three conditions of *too_cold, just_right,* and *too_hot.* The process section of the code uses the IF, ELSIF, and ELSE to identify the range in which the temperature lies and assigns the correct bit pattern to status. In the last three statements, each bit of *status* is connected to the correct output port bit.

```
ENTITY fig4_59  IS
PORT(digital_value:IN INTEGER RANGE 0 TO 15;    -- declare 4-bit input
     too_cold, just_right, too_hot :OUT BIT);
END fig4_59 ;

ARCHITECTURE howhot OF fig4_59  IS
SIGNAL status   :BIT_VECTOR (2 downto 0);
BEGIN
   PROCESS (digital_value)
      BEGIN
         IF (digital_value <= 8) THEN status <= "100";
         ELSIF (digital_value > 8 AND digital_value < 11) THEN
               status <= "010";
         ELSE  status <= "001";
         END IF;
   END PROCESS ;
   too_cold    <= status(2);          -- assign status bits to output
   just_right  <= status(1);
   too_hot     <= status(0);
END howhot;
```

FIGURE 4-59 Temperature range example in VHDL using ELSIF.

CASE

One more important control structure is useful for choosing actions based on current conditions. It is called by various names, depending on the programming language, but it nearly always involves the word **CASE**. This construct determines the value of an expression or object and then goes through a list of possible values (cases) for the expression or object being evaluated. Each case has a list of actions that should take place. A CASE construct is different from an IF/ELSIF because a case correlates one unique value of an object with a set of actions. Recall that an IF/ELSIF correlates a set of actions with a true statement. There can be only one match for a CASE statement. An IF/ELSIF can have more than one statement that is true, but will THEN perform the action associated with the first true statement it evaluates.

Another important point in the examples that follow is the need to combine several independent variables into a set of bits, called a bit vector. Recall that this action of linking several bits in a particular order is called *concatenation*. It allows us to consider the bit pattern as an ordered group.

AHDL

CASE USING AHDL

The AHDL example in Figure 4-60 demonstrates a case construct implementing the circuit of Figure 4-9 (see also Table 4-3). It uses individual bits as its inputs. In the first statement after BEGIN, these bits are concatenated and assigned to the intermediate variable called *status*. The CASE statement evaluates the variable *status* and finds the bit pattern (following the keyword WHEN) that matches the value of *status*. It then performs the action described following =>. In this example, it simply assigns logic 0 to the output for each of the three specified cases. All *other* cases result in a logic 1 on the output.

FIGURE 4-60 Figure 4-9 represented in AHDL.

```
SUBDESIGN fig4_60
(
   p, q, r          :INPUT;      -- define inputs to block
   s                :OUTPUT;     -- define outputs
)
VARIABLE
  status[2..0]      :NODE;
BEGIN
   status[]= (p, q, r);  -- link input bits in order
   CASE status[] IS
      WHEN b"100"       => s = GND;
      WHEN b"101"       => s = GND;
      WHEN b"110"       => s = GND;
      WHEN OTHERS       => s = VCC;
   END CASE;
END;
```

CASE USING VHDL

The VHDL example in Figure 4-61 demonstrates the case construct implementing the circuit of Figure 4-9 (see also Table 4-3). It uses individual bits as its inputs. In the first statement after BEGIN, these bits are concatenated and assigned to the intermediate variable called *status* using the & operator. The CASE statement evaluates the variable status and finds the bit pattern (following the keyword WHEN) that matches the value of *status*. It then performs the action described following =>. In this simple example, it merely assigns logic 0 to the output for each of the three specified cases. All *other* cases result in a logic 1 on the output.

FIGURE 4-61 Figure 4-9 represented in VHDL.

```
ENTITY fig4_61 IS
PORT( p, q, r       :IN bit;                  --declare 3 bits input
      s             :OUT BIT);
END fig4_61;

ARCHITECTURE copy OF fig4_61 IS
SIGNAL status       :BIT_VECTOR (2 downto 0);
BEGIN
   status <= p & q & r;                       --link bits in order.
   PROCESS (status)
      BEGIN
         CASE status IS
            WHEN "100" =>  s <= '0';
            WHEN "101" =>  s <= '0';
            WHEN "110" =>  s <= '0';
            WHEN OTHERS => s <= '1';
         END CASE;
      END PROCESS;
END copy;
```

EXAMPLE 4-34

A coin detector in a vending machine accepts quarters, dimes, and nickels and activates the corresponding digital signal (Q, D, N) only when the correct coin is present. It is physically impossible for multiple coins to be present at the same time. A digital circuit must use the Q, D, and N signals as inputs and produce a binary number representing the value of the coin as shown in Figure 4-62. Write the AHDL and VHDL code.

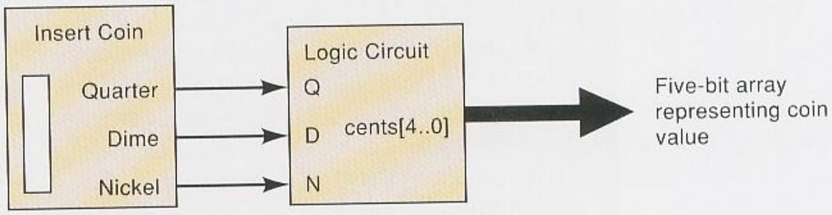


FIGURE 4-62 A coin detector circuit for a vending machine.

Solution

This is an ideal application of the CASE construct to describe the correct operation. The outputs must be declared as five-bit numbers in order to represent up to 25 cents. Figure 4-63 shows the AHDL solution and Figure 4-64 shows the VHDL solution.

AHDL

```
SUBDESIGN   fig4_63
(
   q, d, n         :INPUT;      -- define quarter, dime, nickel
   cents[4..0]     :OUTPUT;     -- define binary value of coins
)
BEGIN
   CASE (q, d, n) IS            -- group coins in an ordered set
      WHEN b"001" => cents[] = 5;
      WHEN b"010" => cents[] = 10;
      WHEN b"100" => cents[] = 25;
      WHEN others => cents[] = 0;
   END CASE;
END;
```

FIGURE 4-63 An AHDL coin detector.

VHDL

```
ENTITY   fig4_64 IS
PORT( q, d, n:IN BIT;                        -- quarter, dime, nickel
      cents  :OUT INTEGER RANGE 0 TO 25);    -- binary value of coins
END fig4_64;
ARCHITECTURE detector of fig4_64 IS
   SIGNAL   coins :BIT_VECTOR(2 DOWNTO 0);-- group the coin sensors
   BEGIN
      coins <= (q & d & n);                  -- assign sensors to group
      PROCESS (coins)
         BEGIN
            CASE (coins) IS
               WHEN "001"   => cents <= 5;
               WHEN "010"   => cents <= 10;
               WHEN "100"   => cents <= 25;
               WHEN others => cents <= 0;
            END CASE;
         END PROCESS;
   END detector;
```

FIGURE 4-64 A VHDL coin detector.

REVIEW QUESTIONS

1. Which control structure decides to do or not to do?
2. Which control structure decides to do this or to do that?
3. Which control structure(s) decides which one of several different actions to take?
4. Declare an input named *count* that can represent a numeric quantity as big as 205. Use AHDL or VHDL.

SUMMARY

1. The two general forms for logic expressions are the sum-of-products form and the product-of-sums form.
2. One approach to the design of a combinatorial logic circuit is to (1) construct its truth table, (2) convert the truth table to a sum-of-products expression, (3) simplify the expression using Boolean algebra or K mapping, (4) implement the final expression.
3. The K map is a graphical method for representing a circuit's truth table and generating a simplified expression for the circuit output.
4. An exclusive-OR circuit has the expression $x = A\overline{B} + \overline{A}B$. Its output x will be HIGH only when inputs A and B are at opposite logic levels.
5. An exclusive-NOR circuit has the expression $x = \overline{A}\,\overline{B} + AB$. Its output x will be HIGH only when inputs A and B are at the same logic level.
6. Each of the basic gates (AND, OR, NAND, NOR) can be used to enable or disable the passage of an input signal to its output.
7. The main digital IC families are the TTL and CMOS families. Digital ICs are available in a wide range of complexities (gates per chip), from the basic to the high-complexity logic functions.
8. To perform basic troubleshooting requires—at a minimum—an understanding of circuit operation, a knowledge of the types of possible faults, a complete logic-circuit connection diagram, and a logic probe.
9. A programmable logic device (PLD) is an IC that contains a large number of logic gates whose interconnections can be programmed by the user to generate the desired logic relationship between inputs and outputs.
10. To program a PLD, you need a development system that consists of a computer, PLD development software, and a programmer fixture that does the actual programming of the PLD chip.
11. The Altera system allows convenient hierarchical design techniques using any form of hardware description.
12. The type of data objects must be specified so that the HDL compiler knows the range of numbers to be represented.
13. Truth tables can be entered directly into the source file using the features of HDL.
14. Logical control structures such as IF, ELSE, and CASE can be used to describe the operation of a logic circuit, making the code and the problem's solution much more straightforward.

IMPORTANT TERMS

sum-of-products (SOP)
product-of-sums (POS)
Karnaugh map (K map)
looping
don't-care condition
exclusive-OR (XOR)
exclusive-NOR (XNOR)
parity generator
parity checker
enable/disable
dual-in-line package (DIP)
SSI, MSI, LSI, VLSI, ULSI, GSI
transistor-transistor logic (TTL)
complementary metal-oxide-semiconductor (CMOS)
indeterminate
floating
logic probe
contention
programmer
ZIF socket
JEDEC
JTAG
hierarchical design
top-down
test vectors
literals
bit array
bit vector
BIT_VECTOR
index
integer
objects
libraries
macrofunction
STD_LOGIC
STD_LOGIC_VECTOR
concatenate
selected signal assignment
decision control structure
concurrent
sequential
IF/THEN
ELSE
PROCESS
sensitivity list
ELSIF
CASE

PROBLEMS

SECTIONS 4-2 AND 4-3

B 4-1.*Simplify the following expressions using Boolean algebra.

(a) $x = ABC + \overline{A}C$
(b) $y = (Q + R)(\overline{Q} + \overline{R})$
(c) $w = ABC + A\overline{B}C + \overline{A}$
(d) $q = \overline{RST}(\overline{R + S + T})$
(e) $x = \overline{A}\,\overline{B}\,\overline{C} + \overline{A}BC + ABC + A\overline{B}\,\overline{C} + A\overline{B}C$
(f) $z = (B + \overline{C})(\overline{B} + C) + \overline{\overline{A} + B + \overline{C}}$
(g) $y = (\overline{C + D}) + \overline{A}C\overline{D} + A\overline{B}\,\overline{C} + \overline{A}\,\overline{B}CD + AC\overline{D}$
(h) $x = AB(\overline{\overline{C}D}) + \overline{A}BD + \overline{B}\,\overline{C}\,\overline{D}$

B 4-2. Simplify the circuit of Figure 4-65 using Boolean algebra.

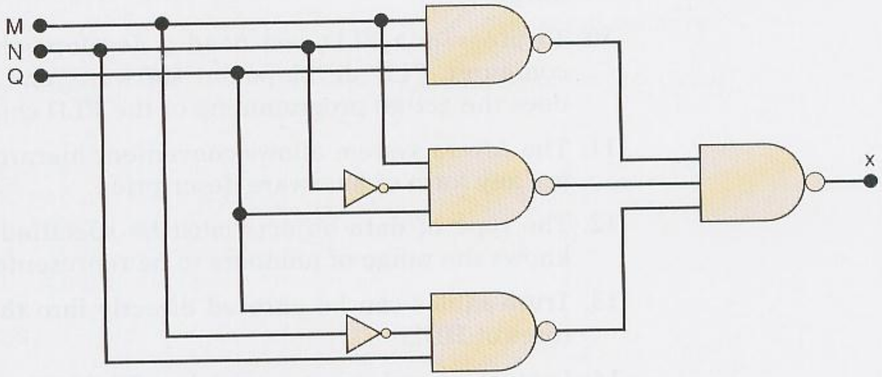


FIGURE 4-65 Problems 4-2 and 4-3.

---

*Answers to problems marked with an asterisk can be found in the back of the text.

B 4-3.* Change each gate in Problem 4-2 to a NOR gate, and simplify the circuit using Boolean algebra.

SECTION 4-4

B, D 4-4.* Design the logic circuit corresponding to the truth table shown in Table 4-11.

TABLE 4-11

A	B	C	x
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

B, D 4-5. Design a logic circuit whose output is HIGH *only* when a majority of inputs A, B, and C are LOW.

D 4-6. A manufacturing plant needs to have a horn sound to signal quitting time. The horn should be activated when either of the following conditions is met:

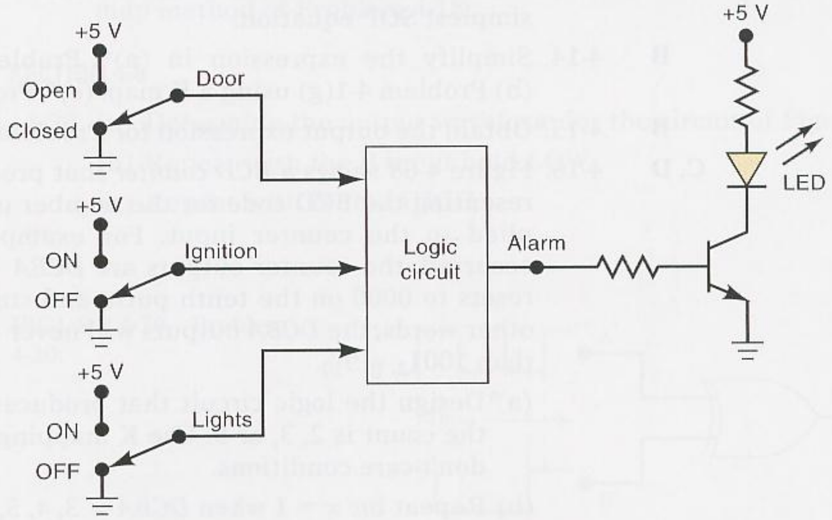
1. It's after 5 o'clock and all machines are shut down.
2. It's Friday, the production run for the day is complete, and all machines are shut down.

Design a logic circuit that will control the horn. (*Hint:* Use four logic input variables to represent the various conditions; for example, input A will be HIGH only when the time of day is 5 o'clock or later.)

D 4-7.* A four-bit binary number is represented as $A_3A_2A_1A_0$, where A_3, A_2, A_1, and A_0 represent the individual bits and A_0 is equal to the LSB. Design a logic circuit that will produce a HIGH output whenever the binary number is greater than 0010 and less than 1000.

D 4-8. Figure 4-66 shows a diagram for an automobile alarm circuit used to detect certain undesirable conditions. The three switches are used to

FIGURE 4-66 Problem 4-8.



indicate the status of the door by the driver's seat, the ignition, and the headlights, respectively. Design the logic circuit with these three switches as inputs so that the alarm will be activated whenever either of the following conditions exists:

- The headlights are on while the ignition is off.
- The door is open while the ignition is on.

4-9.*Implement the circuit of Problem 4-4 using all NAND gates.

4-10. Implement the circuit of Problem 4-5 using all NAND gates.

SECTION 4-5

B 4-11. Determine the minimum expression for each K map in Figure 4-67. Pay particular attention to step 5 for the map in (a).

	$\overline{C}\overline{D}$	$\overline{C}D$	CD	$C\overline{D}$
$\overline{A}\overline{B}$	1	1	1	1
$\overline{A}B$	1	1	0	0
AB	0	0	0	1
$A\overline{B}$	0	0	1	1

(a)*

	$\overline{C}\overline{D}$	$\overline{C}D$	CD	$C\overline{D}$
$\overline{A}\overline{B}$	1	0	1	1
$\overline{A}B$	1	0	0	1
AB	0	0	0	0
$A\overline{B}$	1	0	1	1

(b)

	$\overline{C}$	C
$\overline{A}\overline{B}$	1	1
$\overline{A}B$	0	0
AB	1	0
$A\overline{B}$	1	X

(c)

FIGURE 4-67 Problem 4-11.

B 4-12. For the truth table below, create a 2 × 2 K map, group terms, and simplify. Then look at the truth table again to see if the expression is true for every entry in the table.

A	B	y
0	0	1
0	1	1
1	0	0
1	1	0

B 4-13. Starting with the truth table in Table 4-11, use a K map to find the simplest SOP equation.

B 4-14. Simplify the expression in (a)* Problem 4-1(e) using a K map. (b) Problem 4-1(g) using a K map. (c)* Problem 4-1(h) using a K map.

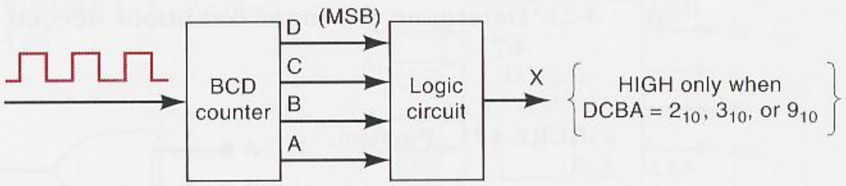
B 4-15.*Obtain the output expression for Problem 4-7 using a K map.

C, D 4-16. Figure 4-68 shows a *BCD counter* that produces a four-bit output representing the BCD code for the number of pulses that have been applied to the counter input. For example, after four pulses have occurred, the counter outputs are $DCBA = 0100_2 = 4_{10}$. The counter resets to 0000 on the tenth pulse and starts counting over again. In other words, the *DCBA* outputs will never represent a number greater than $1001_2 = 9_{10}$.

(a)*Design the logic circuit that produces a HIGH output whenever the count is 2, 3, or 9. Use K mapping and take advantage of the don't-care conditions.

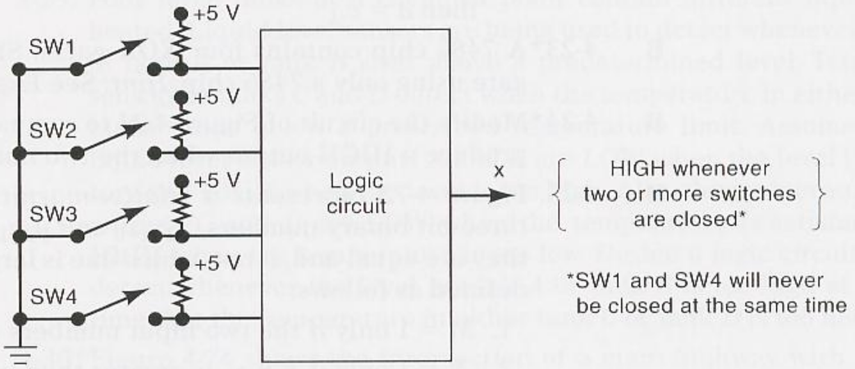
(b) Repeat for $x = 1$ when $DCBA = 3, 4, 5, 8$.

FIGURE 4-68 Problem 4-16.


D (MSB)
C
B
A
BCD counter
Logic circuit
X
HIGH only when DCBA = 2_{10}, 3_{10}, or 9_{10}

D 4-17.* Figure 4-69 shows four switches that are part of the control circuitry in a copy machine. The switches are at various points along the path of the copy paper as the paper passes through the machine. Each switch is normally open, and as the paper passes over a switch, the switch closes. It is impossible for switches SW1 and SW4 to be closed at the same time. Design the logic circuit to produce a HIGH output whenever *two or more* switches are closed at the same time. Use K mapping and take advantage of the don't-care conditions.

FIGURE 4-69 Problem 4-17.


+5 V
SW1
+5 V
SW2
+5 V
SW3
+5 V
SW4
Logic circuit
x
HIGH whenever two or more switches are closed*
*SW1 and SW4 will never be closed at the same time

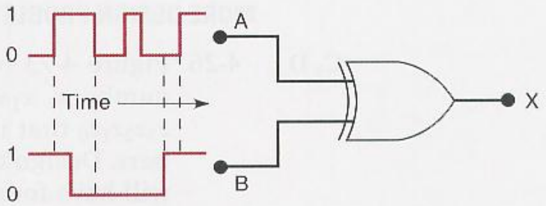
B 4-18. Example 4-3 demonstrated algebraic simplification. Step 3 resulted in the SOP equation $z = \bar{A}\,\bar{B}C + \bar{A}C\bar{D} + \bar{A}B\bar{C}\,\bar{D} + A\bar{B}C$. Use a K map to prove that this equation can be simplified further than the answer shown in the example.

C 4-19. Use Boolean algebra to arrive at the same result obtained by the K map method of Problem 4-18.

SECTION 4-6

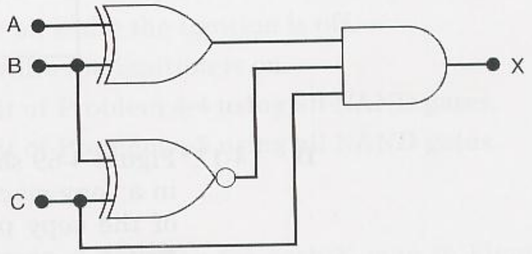
B 4-20. (a) Determine the output waveform for the circuit of Figure 4-70.
(b) Repeat with the *B* input held LOW.
(c) Repeat with *B* held HIGH.

FIGURE 4-70 Problem 4-20.


1
0
A
Time
X
1
0
B

B 4-21.*Determine the input conditions needed to produce $x = 1$ in Figure 4-71.

FIGURE 4-71 Problem 4-21.



B 4-22. Design a circuit that produces a HIGH out only when all three inputs are the same level.

(a) Use a truth table and K map to produce the SOP solution.

(b) Use two-input XOR and other gates to find a solution. (*Hint:* Recall the transitive property from algebra. . . if $a = b$ and $b = c$ then $a = c$.)

B 4-23.*A 7486 chip contains four XOR gates. Show how to make an XNOR gate using only a 7486 chip. *Hint:* See Example 4-16.

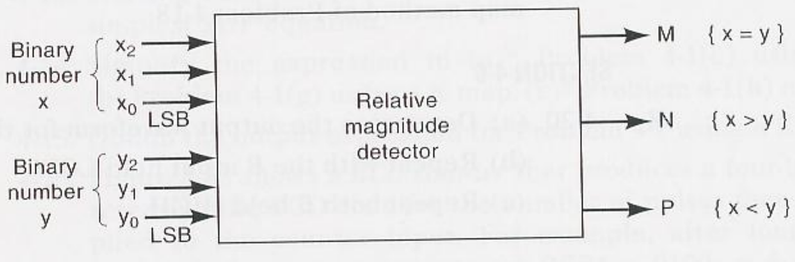
B 4-24.*Modify the circuit of Figure 4-23 to compare two four-bit numbers and produce a HIGH output when the two numbers match exactly.

B 4-25. Figure 4-72 represents a *relative-magnitude detector* that takes two three-bit binary numbers, $x_2x_1x_0$ and $y_2y_1y_0$, and determines whether they are equal and, if not, which one is larger. There are three outputs, defined as follows:

1. $M = 1$ only if the two input numbers are equal.
2. $N = 1$ only if $x_2x_1x_0$ is greater than $y_2y_1y_0$.
3. $P = 1$ only if $y_2y_1y_0$ is greater than $x_2x_1x_0$.

Design the logic circuitry for this detector. The circuit has *six* inputs and *three* outputs and is therefore much too complex to handle using the truth-table approach. Refer to Example 4-17 as a hint about how you might start to solve this problem.

FIGURE 4-72 Problem 4-25.



MORE DESIGN PROBLEMS

C, D 4-26.*Figure 4-73 represents a multiplier circuit that takes two-bit binary numbers, x_1x_0 and y_1y_0, and produces an output binary number $z_3z_2z_1z_0$ that is equal to the arithmetic product of the two input numbers. Design the logic circuit for the multiplier. (*Hint:* The logic circuit will have four inputs and four outputs.)

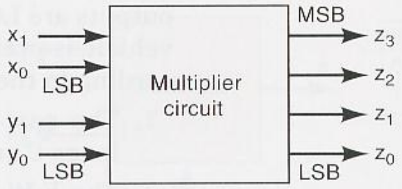


FIGURE 4-73 Problem 4-26.

D 4-27. A BCD code is being transmitted to a remote receiver. The bits are A_3, A_2, A_1, and A_0, with A_3 as the MSB. The receiver circuitry includes a *BCD error detector* circuit that examines the received code to see if it is a legal BCD code (i.e., ≤1001). Design this circuit to produce a HIGH for any error condition.

D 4-28.* Design a logic circuit whose output is HIGH whenever *A* and *B* are both HIGH as long as *C* and *D* are either both LOW or both HIGH. Try to do this without using a truth table. Then check your result by constructing a truth table from your circuit to see if it agrees with the problem statement.

D 4-29. Four large tanks at a chemical plant contain different liquids being heated. Liquid-level sensors are being used to detect whenever the level in tank *A* or tank *B* rises above a predetermined level. Temperature sensors in tanks *C* and *D* detect when the temperature in either of these tanks drops below a prescribed temperature limit. Assume that the liquid-level sensor outputs *A* and *B* are LOW when the level is satisfactory and HIGH when the level is too high. Also, the temperature-sensor outputs *C* and *D* are LOW when the temperature is satisfactory and HIGH when the temperature is too low. Design a logic circuit that will detect whenever the level in tank *A* or tank *B* is too high at the same time that the temperature in either tank *C* or tank *D* is too low.

C, D 4-30.* Figure 4-74 shows the intersection of a main highway with a secondary access road. Vehicle-detection sensors are placed along lanes *C* and *D* (main road) and lanes *A* and *B* (access road). These sensor

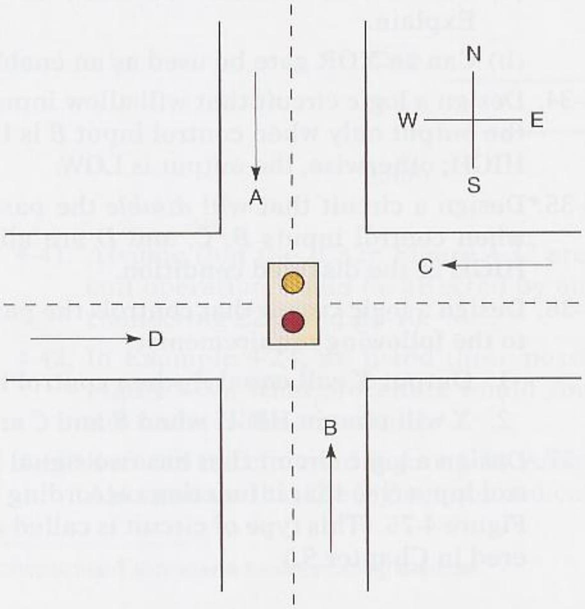


FIGURE 4-74 Problem 4-30.

outputs are LOW (0) when no vehicle is present and HIGH (1) when a vehicle is present. The intersection traffic light is to be controlled according to the following logic:

1. The east-west (E-W) traffic light will be green whenever *both* lanes C and D are occupied.
2. The E-W light will be green whenever *either* C or D is occupied but lanes A and B are not *both* occupied.
3. The north-south (N-S) light will be green whenever *both* lanes A and B are occupied but C and D are not *both* occupied.
4. The N-S light will also be green when *either* A or B is occupied while C and D are *both* vacant.
5. The E-W light will be green when *no* vehicles are present.

Using the sensor outputs A, B, C, and D as inputs, design a logic circuit to control the traffic light. There should be two outputs, N-S and E-W, that go HIGH when the corresponding light is to be *green.* Simplify the circuit as much as possible and show *all* steps.

SECTION 4-7

D 4-31. Redesign the parity generator and checker of Figure 4-25 to (a) operate using odd parity. (*Hint:* What is the relationship between an odd-parity bit and an even-parity bit for the same set of data bits?) (b) Operate on eight data bits.

SECTION 4-8

B 4-32. (a) Under what conditions will an OR gate allow a logic signal to pass through to its output unchanged?

(b) Repeat (a) for an AND gate.

(c) Repeat for a NAND gate.

(d) Repeat for a NOR gate.

B 4-33.* (a) Can an INVERTER be used as an enable/disable circuit? Explain.

(b) Can an XOR gate be used as an enable/disable circuit? Explain.

D 4-34. Design a logic circuit that will allow input signal A to pass through to the output only when control input B is LOW while control input C is HIGH; otherwise, the output is LOW.

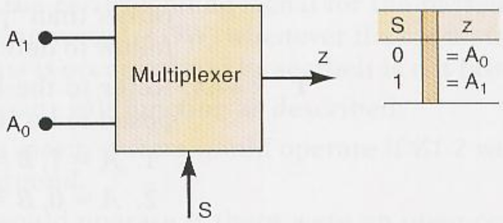
D 4-35.* Design a circuit that will *disable* the passage of an input signal only when control inputs B, C, and D are all HIGH; the output is to be HIGH in the disabled condition.

D 4-36. Design a logic circuit that controls the passage of a signal A according to the following requirements:

1. Output X will equal A when control inputs B and C are the same.
2. X will remain HIGH when B and C are different.

D 4-37. Design a logic circuit that has two signal inputs, A_1 and A_0, and a control input S so that it functions according to the requirements given in Figure 4-75. (This type of circuit is called a *multiplexer* and will be covered in Chapter 9.)

FIGURE 4-75 Problem 4-37.



S	z
0	= A_0
1	= A_1

D 4-38.* Use K mapping to design a circuit to meet the requirements of Example 4-17. Compare this circuit with the solution in Figure 4-23. This points out that the K-map method cannot take advantage of the XOR and XNOR gate logic. The designer must be able to determine when these gates are applicable.

SECTIONS 4-9 TO 4-13

T* 4-39. (a) A technician testing a logic circuit sees that the output of a particular INVERTER is stuck LOW while its input is pulsing. List as many possible reasons as you can for this faulty operation.

(b) Repeat part (a) for the case where the INVERTER output is stuck at an indeterminate logic level.

T 4-40.* The signals shown in Figure 4-76 are applied to the inputs of the circuit of Figure 4-32. Suppose that there is an internal open circuit at Z1-4.

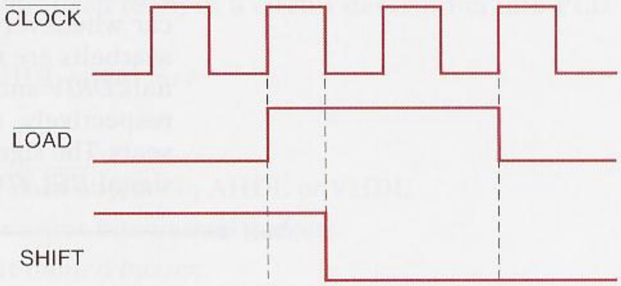
(a) What will a logic probe indicate at Z1-4?

(b) What dc voltage reading would you expect at Z1-4? (Remember that the ICs are TTL.)

(c) Sketch what you think the $\overline{CLKOUT}$ and $\overline{SHIFTOUT}$ signals will look like.

(d) Instead of the open at Z1-4, suppose that pins 9 and 10 of Z2 are internally shorted. Sketch the probable signals at Z2-10, $\overline{CLOCKOUT}$, and $\overline{SHIFTOUT}$.

FIGURE 4-76 Problem 4-40.



T 4-41. Assume that the ICs in Figure 4-32 are CMOS. Describe how the circuit operation would be affected by an open circuit in the conductor connecting Z2-2 and Z2-10.

T 4-42. In Example 4-24, we listed three possible faults for the situation of Figure 4-36. What procedure would you follow to determine which of the faults is the actual one?

T 4-43.* Refer to the circuit of Figure 4-38. Assume that the devices are CMOS. Also assume that the logic probe indication at Z2-3 is "indeterminate" rather than "pulsing." List the possible faults, and write a procedure to follow to determine the actual fault.

---

*Recall that T indicates a troubleshooting exercise.

T 4-44.* Refer to the logic circuit of Figure 4-41. Recall that output Y is supposed to be HIGH for either of the following conditions:

1. $A = 1, B = 0$, regardless of C
2. $A = 0, B = 1, C = 1$

When testing the circuit, the technician observes that Y goes HIGH only for the first condition but stays LOW for all other input conditions. Consider the following list of possible faults. For each one, write yes or no to indicate whether or not it could be the actual fault. Explain your reasoning for each no response.

(a) An internal short to ground at Z2-13
(b) An open circuit in the connection to Z2-13
(c) An internal short to V_{CC} at Z2-11
(d) An open circuit in the V_{CC} connection to Z2
(e) An internal open circuit at Z2-9
(f) An open in the connection from Z2-11 to Z2-9
(g) A solder bridge between pins 6 and 7 of Z2

T 4-45. Develop a procedure for isolating the fault that is causing the malfunction described in Problem 4-44.

T 4-46.* Assume that the gates in Figure 4-41 are all CMOS. When the technician tests the circuit, he finds that it operates correctly except for the following conditions:

1. $A = 1, B = 0, C = 0$
2. $A = 0, B = 1, C = 1$

For these conditions, the logic probe indicates indeterminate levels at Z2-6, Z2-11, and Z2-8. What do you think is the probable fault in the circuit? Explain your reasoning.

T 4-47. Figure 4-77 is a combinational logic circuit that operates an alarm in a car whenever the driver and/or passenger seats are occupied and the seatbelts are not fastened when the car is started. The active-HIGH signals *DRIV* and *PASS* indicate the presence of the driver and passenger, respectively, and are taken from pressure-actuated switches in the seats. The signal *IGN* is active-HIGH when the ignition switch is on. The signal $\overline{BELTD}$ is active-LOW and indicates that the driver's seatbelt is

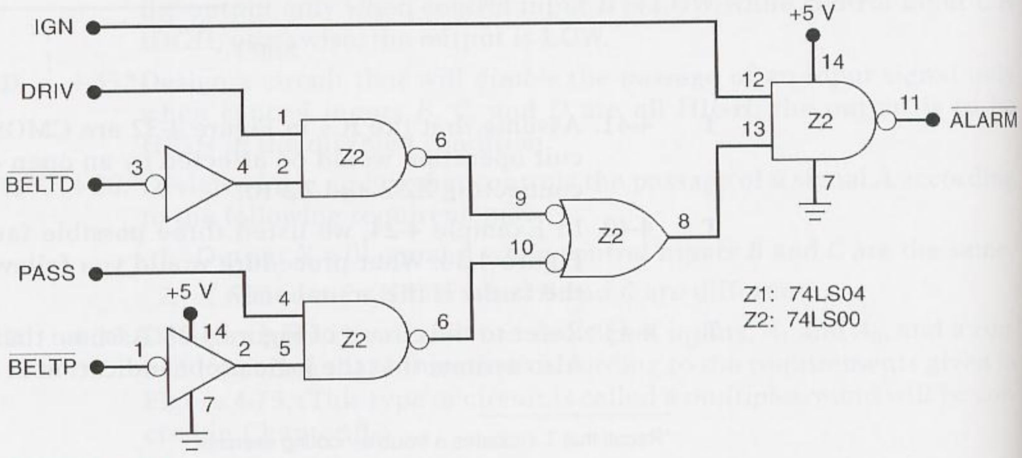


FIGURE 4-77 Problems 4-47, 4-48, and 4-49.

unfastened; $\overline{BELTP}$ is the corresponding signal for the passenger seatbelt. The alarm will be activated (LOW) whenever the car is started and either of the front seats is occupied and its seatbelt is not fastened.

(a) Verify that the circuit will function as described.

(b) Describe how this alarm system would operate if Z1-2 were internally shorted to ground.

(c) Describe how it would operate if there were an open connection from Z2-6 to Z2-10.

T 4-48.* Suppose that the system of Figure 4-77 is functioning so that the alarm is activated as soon as the driver and/or passenger are seated and the car is started, regardless of the status of the seatbelts. What are the possible faults? What procedure would you follow to find the actual fault?

T 4-49.* Suppose that the alarm system of Figure 4-77 is operating so that the alarm goes on continuously as soon as the car is started, regardless of the state of the other inputs. List the possible faults and write a procedure to isolate the fault.

DRILL QUESTIONS ON PLDs (50 THROUGH 55)

4-50.* *True or false:*

(a) Top-down design begins with an overall description of the entire system and it specifications.

(b) A JEDEC file can be used as the input file for a programmer.

(c) If an input file compiles with no errors, it means the PLD circuit will work correctly.

(d) A compiler can interpret code in spite of syntax errors.

(e) Test vectors are used to simulate and test a device.

H, B 4-51. What are the % characters used for in the AHDL design file?

H, B 4-52. How are comments indicated in a VHDL design file?

B 4-53. What is a ZIF socket?

B 4-54.* Name three entry modes used to input a circuit description into PLD development software.

B 4-55. What do JEDEC and HDL stand for?

SECTION 4-15

H, B 4-56. Declare the following data objects in AHDL or VHDL.

(a)* An array of eight output bits named *gadgets*.

(b) A single-output bit named *buzzer*.

(c) A 16-bit numeric input port named *altitude*.

(d) A single, intermediate bit within a hardware description file named *wire2*.

H, B 4-57. Express the following literal numbers in hex, binary, and decimal using the syntax of AHDL or VHDL.

(a)* 152_{10}

(b) 1001010100_2

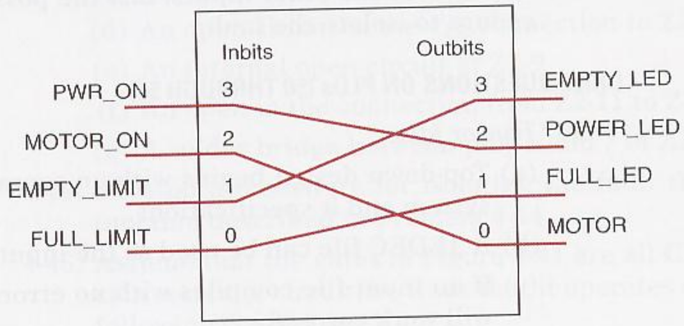
(c) $3C4_{16}$

H, B 4-58.* The following similar I/O definition is given for AHDL and VHDL. Write four concurrent assignment statements that will connect the inputs to the outputs as shown in Figure 4-78.

FIGURE 4-78 Problem 4-58.

```
SUBDESIGN hw
(
     inbits[3..0]    :INPUT;
     outbits[3..0]   :OUTPUT;
)
```

```
ENTITY hw IS
PORT (
     inbits      :IN BIT_VECTOR (3 downto 0);
     outbits     :OUT BIT_VECTOR (3 downto 0)
     );
END hw;
```



SECTION 4-16

H, D 4-59. Modify the AHDL truth table of Figure 4-50 to implement $AB + A\overline{C} + \overline{A}B$.

H, D 4-60.* Modify the AHDL design in Figure 4-54 so that $z = 1$ only when the digital value is less than 1010_2.

H, D 4-61. Modify the VHDL truth table of Figure 4-51 to implement $AB + A\overline{C} + \overline{A}B$.

H, D 4-62.* Modify the VHDL design in Figure 4-55 so that $z = 1$ only when the digital value is less than 1010_2.

H, B 4-63. Modify the code of (a) Figure 4-54 or (b) Figure 4-55 such that the output z is LOW only when digital_value is between 6 and 11 (inclusive).

H, D 4-64. Modify (a) the AHDL design in Figure 4-60 to implement Table 4-1. (b) the VHDL design in Figure 4-61 to implement Table 4-1.

H, D 4-65.* Write the hardware description design file Boolean equation to implement Example 4-9.

4-66. Write the hardware description design file Boolean equation to implement a four-bit parity generator as shown in Figure 4-25(a).

DRILL QUESTION

B 4-67. Define each of the following terms.

(a) Karnaugh map

(b) Sum-of-products form

(c) Parity generator

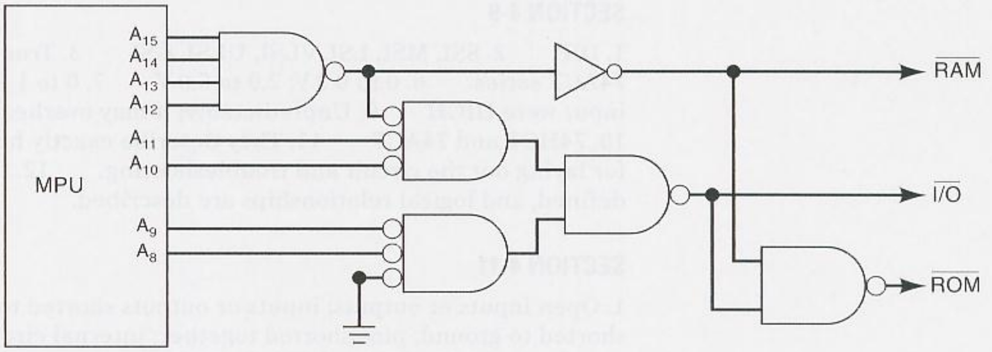
(d) Octet

(e) Enable circuit
(f) Don't-care condition
(g) Floating input
(h) Indeterminate voltage level
(i) Contention
(j) PLD
(k) TTL
(l) CMOS

MICROCOMPUTER APPLICATIONS

C 4-68. In a microcomputer, the microprocessor unit (MPU) is always communicating with one of the following: (1) random-access memory (RAM), which stores programs and data that can be readily changed; (2) read-only memory (ROM), which stores programs and data that never change; and (3) external input/output (I/O) devices such as keyboards, video displays, printers, and disk drives. As it is executing a program, the MPU will generate an address code that selects which type of device (RAM, ROM, or I/O) it wants to communicate with. Figure 4-79 shows a typical arrangement where the MPU outputs an eight-bit address code A_{15} through A_8. Actually, the MPU outputs a 16-bit address code, but the low-order bits A_7 through A_0 are not used in the device selection process. The address code is applied to a logic circuit that uses it to generate the device select signals: $\overline{RAM}$, $\overline{ROM}$, and $\overline{I/O}$.

FIGURE 4-79 Problem 4-68.



Analyze this circuit and determine the following.

(a)*The range of addresses A_{15} through A_8 that will activate $\overline{\text{RAM}}$
(b) The range of addresses that activate $\overline{\text{I/O}}$
(c) The range of addresses that activate $\overline{\text{ROM}}$

Express the addresses in binary and hexadecimal. For example, the answer to (a) is A_{15} to $A_8 = 00000000_2$ to $11101111_2 = 00_{16}$ to EF_{16}.

C, D 4-69. In some microcomputers, the MPU can be *disabled* for short periods of time while another device controls the RAM, ROM, and I/O. During these intervals, a special control signal ($\overline{DMA}$) is activated by the MPU and is used to disable (deactivate) the device select logic so that the $\overline{\text{RAM}}$, $\overline{\text{ROM}}$, and $\overline{\text{I/O}}$ are all in their inactive state. Modify the circuit of Figure 4-79 so that $\overline{\text{RAM}}$, $\overline{\text{ROM}}$, and $\overline{\text{I/O}}$ will be deactivated whenever the signal *DMA* is active, regardless of the state of the address code.

ANSWERS TO SECTION REVIEW QUESTIONS

SECTION 4-1

1. Only (a) 2. Only (c)

SECTION 4-3

1. Expression (b) is not in sum-of-products form because of the inversion sign over both the C and D variables (i.e., the $A\overline{CD}$ term). Expression (c) is not in sum-of-products form because of the $(M + \overline{N})P$ term. 3. $x = \overline{A} + \overline{B} + \overline{C}$

SECTION 4-4

1. $x = \overline{A}\,\overline{B}\,\overline{C}D + \overline{A}\,\overline{B}C\overline{D} + \overline{A}B\overline{C}\,\overline{D}$ 2. Eight

SECTION 4-5

1. $x = AB + AC + BC$ 2. $x = A + BCD$ 3. $S = \overline{P} + QR$ 4. An input condition for which there is no specific required output condition; i.e., we are free to make it 0 or 1.

SECTION 4-6

2. A constant LOW 3. No; the available XOR gate can be used as an INVERTER by connecting one of its inputs to a constant HIGH (see Example 4-16).

SECTION 4-8

1. $x = \overline{A(B \oplus C)}$ 2. OR, NAND 3. NAND, NOR

SECTION 4-9

1. DIP 2. SSI, MSI, LSI, VLSI, ULSI, GSI 3. True 4. True 5. 40, 74AC, 74ACT series 6. 0 to 0.8 V; 2.0 to 5.0 V 7. 0 to 1.5 V; 3.5 to 5.0 V 8. As if the input were HIGH 9. Unpredictably; it may overheat and be destroyed. 10. 74HCT and 74ACT 11. They describe exactly how to interconnect the chips for laying out the circuit and troubleshooting. 12. Inputs and outputs are defined, and logical relationships are described.

SECTION 4-11

1. Open inputs or outputs; inputs or outputs shorted to V_{CC}; inputs or outputs shorted to ground; pins shorted together; internal circuit failures 2. Pins shorted together 3. For TTL, a LOW; for CMOS, indeterminate 4. Two or more outputs connected together

SECTION 4-12

1. Open signal lines; shorted signal lines; faulty power supply; output loading 2. Broken wires; poor solder connections; cracks or cuts in PC board; bent or broken IC pins; faulty IC sockets 3. ICs operating erratically or not at all 4. Logic level indeterminate

SECTION 4-14

1. Electrically controlled connections are being programmed as open or closed. 2. (4, 1) (2, 2) or (2, 1) (4, 2) 3. (4, 5) (1, 6) or (4, 6) (1, 5)

SECTION 4-15

1. (a) push_buttons[5..0] :INPUT; (b) push_buttons :IN BIT_VECTOR (5 DOWNTO 0), 2. (a) z = push_buttons[5]; (b) z <= push_buttons(5); 3. STD_LOGIC 4. STD_LOGIC_VECTOR

SECTION 4-16

1. (AHDL) omega[] = (x, y, z); (VHDL) omega <= x & y & z; 2. Using the keyword TABLE 3. Using selected signal assignments

SECTION 4-17

1. IF/THEN 2. IF/THEN/ELSE 3. CASE or IF/ELSIF
4. (AHDL) count[7..0] :INPUT; (VHDL) count :IN INTEGER RANGE 0 TO 205